PROGRAMOWANIE MC68K W JĘZYKU ASEMBLERA

©Marek Wnuk

7 grudnia 2000

1 Architektura procesora MC68000

Nasze rozważania na temat architektury mikroprocesora MC68000 rozpoczniemy od przedstawienia ogólnej budowy wewnętrznej komputera. Na tym tle przyjrzymy się dokładniej strukturze wewnętrznej MC68000 próbując dociec, dlaczego przyjęto poszczególne rozwiązania. Omówimy sposoby wewnętrznego reprezentowania danych w mikroprocesorze i tryby adresowania pozwalające na dostęp do tych danych.

1.1 Podstawowe składniki sprzętowe mikrokomputera

W typowych komputerach można z grubsza wydzielić trzy elementy składowe:

- jednostkę centralną,
- pamięć,
- układy wejścia/wyjścia.

Przepływ danych następuje pomiędzy każdą parą wymienionych części składowych komputera. Jednostka centralna steruje pracą obu pozostałych elementów.

1.1.1 Jednostka centralna

Jednostka centralna, zwana procesorem lub CPU, stanowi "mózg" komputera. Wykonuje ona odpowiednie (zlecone jej) operacje. Zlecenie (komenda) może mieć różną postać. Weźmy jako przykład dodawanie dwóch liczb:

```
Z = A + B
```

Można to zapisać w pseudoangielskiej notacji:

```
ADD X to Y and SAVE the result toZ
```

co w uproszczonym symbolicznym zapisie można przedstawić:

```
X + Y -> Z
```

co czytamy: **"dodaj liczbę X do liczby Y i wynik prześlij do Z"**.
Komenda ta, po przetłumaczeniu na zrozumiały dla CPU kod binarny winna spowodować następujące czynności procesora:

- pobranie liczby X do CPU
- pobranie liczby Y do CPU
- dodanie X i Y
- złożenie wyniku w Z

Takie komendy tworzą wewnętrzny język maszyny i są zwane instrukcjami maszynowymi. Składają się z dwóch części:

- kodu operacji (*opcode*) ADD,
- wartości danych do przetwarzania.

Ciąg takich rozkazów maszynowych nazywamy programem w języku maszynowym (*machine language program*). Zestaw wszystkich rozkazów maszynowych, które dany komputer może wykonywać zwie się listą rozkazów procesora (*instruction set*). W zależności od typu procesora moźe się ona zawierać od kilkunastu do paruset instrukcji. Procesor może dostawać się do danych przechowywanych w pamięci na wiele różnych sposobów. Sposoby te zwie się trybami adresowania (*addressing modes*). Ich ilość jest różna dla różnych procesorów. Z punktu widzenia programisty istotne są również sposoby reprezentacji danych dostępne w danym komputerze.

1.1.2 Pamięć

Opisane wcześniej instrukcje maszynowe i związane z nimi dane muszą być przechowywane w pamięci systemu komputerowego. W architekturze von Neumann'a komórki pamięci są identyczne dla instrukcji (programu) i dla danych. Komórka pamięci jest czasem zwana słowem (*word*), a ilość bitów składających się na słowo zwie się rozmiarem słowa (*wordsize*). Każdej komórce pamięci jest jednoznacznie przyporządkowana liczba zwana adresem słowa. Adresy liczy się zazwyczaj od 0 do pewnej górnej granicy, zależnej od wielkości pamięci.

1.1.3 Wykonywanie instrukcji

Wróćmy do przykładu dodawania dwóch liczb. Jak wspomniano, zarówno dane, jak i instrukcje muszą być zakodowane i umieszczone w komórkach pamięci. Rozważmy, jakie komórki są niezbędne do przechowania programu realizującego dodawanie oraz danych bbiorących w nim udział.
Przede wszystkim, kod operacji (ADD) musi zajmować słowo (co najmniej jedno). Aby móc pobrać liczby X i Y, trzeba znać miejsca, w których są one przechowywane, czyli adresy komórek na te liczby zarezerwowanych. Wreszcie, wynik (Z) musi zostać wpisany w określone miejsce w pamięci, co wymaga kolejnego adresu. Widać więc, że kompletny rozkaz maszynowy w naszym prostym przykładzie składać się musi z co najmniej czterech słów:

- kod operacji,
- adres pierwszej liczby,
- adres drugiej liczby,
- adres wyniku.

Oprćz tego w wykonaniu tego rozkazu biorą udział trzy komórki danych przeznaczone na X, Y i Z.
Komórki zawierające kolejne elementy rozkazu umieszczone są w pamięci sekwencyjnie (mają kolejne adresy). Dzięki temu wprowadzenie rejestru zwanego licznikiem rozkazów (PC – *Program Counter*) pozwala uniknąć pamiętania dodatkowo w rozkazie maszynowym adresu następnego rozkazu do wykonania. Nawiasem mówiąc, nigdy nie pojawiły się procesory pracujące z czterema adresami w rozkazie. Zamiast tego PC jest zwiększany po każdym pobraniu elementu rozkazu maszynowego i po jego wykonaniu zawiera adres kodu operacji następnego rozkazu do wykonania. Wróćmy do naszego przykładu:

```
ADD    adr.1   adr.2   adr.3
                                  (adr.1) + (adr.2) -> adr.3
```

Ten zapis uwzględnia fakt, że argumenty operacji dodawania są pobierane z pamięci. Ujęcie adresów w nawiasy oznacza, że w operacji bierze udział nie sam ades, ale zawartość komórki pamięci przezeń wskazywanej: **"dodaj zawartść komórki o adresie adr.1 do zawartości komórki o adresie adr.2 i wynik prześlij do komórki o adresie adr.3"**. Warto zwrócić uwagę, że prawa strona operatora przesyłania **musi być** adresem.

W powyższym przykładzie cykl wykonania rozkazy maszynowego (cykl rozkazowy) wyglądałby następująco:

1. pobranie komórki pamięci o adresie zawartym w *PC* i zwiększenie *PC*;
2. zdekodowanie pobranej wartości jako kodu rozkazu (*ADD*);
3. wykonanie rozkazu *ADD*:
   (a) pobranie adresu pierwszej liczby z komórki wskazywanej przez *PC* i zwiększenie *PC*;
   (b) pobranie adresu drugiej liczby z komórki wskazywanej przez *PC* i zwiększenie *PC*;
   (c) pobranie adresu wyniku z komórki wskazywanej przez *PC* i zwiększenie *PC*;
   (d) pobranie wartości pierwszej liczby z komórki o adresie pierwszym;
   (e) pobranie wartości drugiej liczby z komórki o adresie drugim;
   (f) wykonanie dodawania;
   (g) wpisanie sumy do komórki o adresie trzecim.

Po takim cyklu *PC* wskazuje na następną komórkę w ciągu zawierającym program, która powinna zawierać kod następnego rozkazu maszynowego. Nie ma przeszkód, by cykl został powtórzony, oczywiście dla kolejnych rozkazów programu. W przypadku konieczności zmiany sekwencyjnego sposobu wykonania programu posługujemy się rozkazem skoku (*GOTO*), którego argumentem jest adres komórki pamięci zawierającej kod maszynowy następnego rozkazu. Jego wykonanie polega po prostu na wstawieniu tego adresu do *PC*.
Opisany powyżej sposób pracy procesora dotyczy tzw. maszyn trójadresowych. Łatwo sprawdzić, że bezpośrednie umieszczenie trzech adresów w rozkazie wydłuża kod maszynowy programu. Weźmy za przykład procesor o 16–bitowym słowie i 32–bitowym adresie. Na powyższy rozkaz trzeba zużyć:

```
ADD    adr.1   adr.2   adr.3
 1   +   2   +   2   +   2    =    7
```

słów. Oczywiście, nie wszystkie rozkazy korzystają z trzech adresów. Aby nie marnować miejsca w pamięci, wprowadzono zmienną długość instrukcji. Procesor w chwili dekodowania kodu operacji (punkt 2) określa, ile adresów musi pobrać dla danego typu rozkazu (np. dla *GOTO* tylko jeden).
Maszyny dwuadresowe posiadają rejestr roboczy dla danych, zwany akumulatorem. Sumę dwóch liczb można wtedy umieścić w tym rejestrze, a następnie zapisać w pamięci dodatkowym rozkazem *SAVE*:

```
ADD    adr.1   adr.2          (adr.1) + (adr.2) -> AKUMULATOR
SAVE   adr.3                  (AKUMULATOR) -> adr.3
```

Proszę zwrócić uwagę, że *AKUMULATOR* jest rejestrem, ale jest traktowany jako adres pewnej komórki (w niektórych typach mikroprocesorów akumulatory są rzeczywiście w taki sposób zaimplementowane).
Następny krok to ograniczenie ilości adresów do jednego:

```
LOAD   adr.1                  (adr.1) -> AKUMULATOR
ADD    adr.2                  (AKUMULATOR) + (adr.2) -> AKUMULATOR
SAVE   adr.3                  (AKUMULATOR) -> adr.3
```

co wymaga nowego rozkazu *LOAD*.
Można sprawdzić, że obie powyższe implementacje są pozornie mniej efektywne od trójadresowej (odpowiednio: 8 i 9 słów). Ponieważ jednak najczęściej mamy do czynienia z ciągami operacji wykonywanych na wspólnych argumentach, proporcje te zmieniają się na korzyść. Proszę rozważyć sumowanie N liczb. Sekwencyjny program w maszynie trójadresowej miałby długość $(N-1)\times 7$ słów, a w maszynie jednoadresowej: $3+((N-1)\times 3)+3$. Już dla $N=3$ krótszy jest program dla maszyny jednoadresowej.
Powyższe przykłady mają tylko uzmysłowić problemy doboru architektury jednostki centralnej przed prezentacją konkretnych rozwiązań, przyjętych w mikroprocesorach M68K firmy Motorola, które będziemy omawiać szczegółowo.

1.1.4 Układy wejściowo–wyjściowe

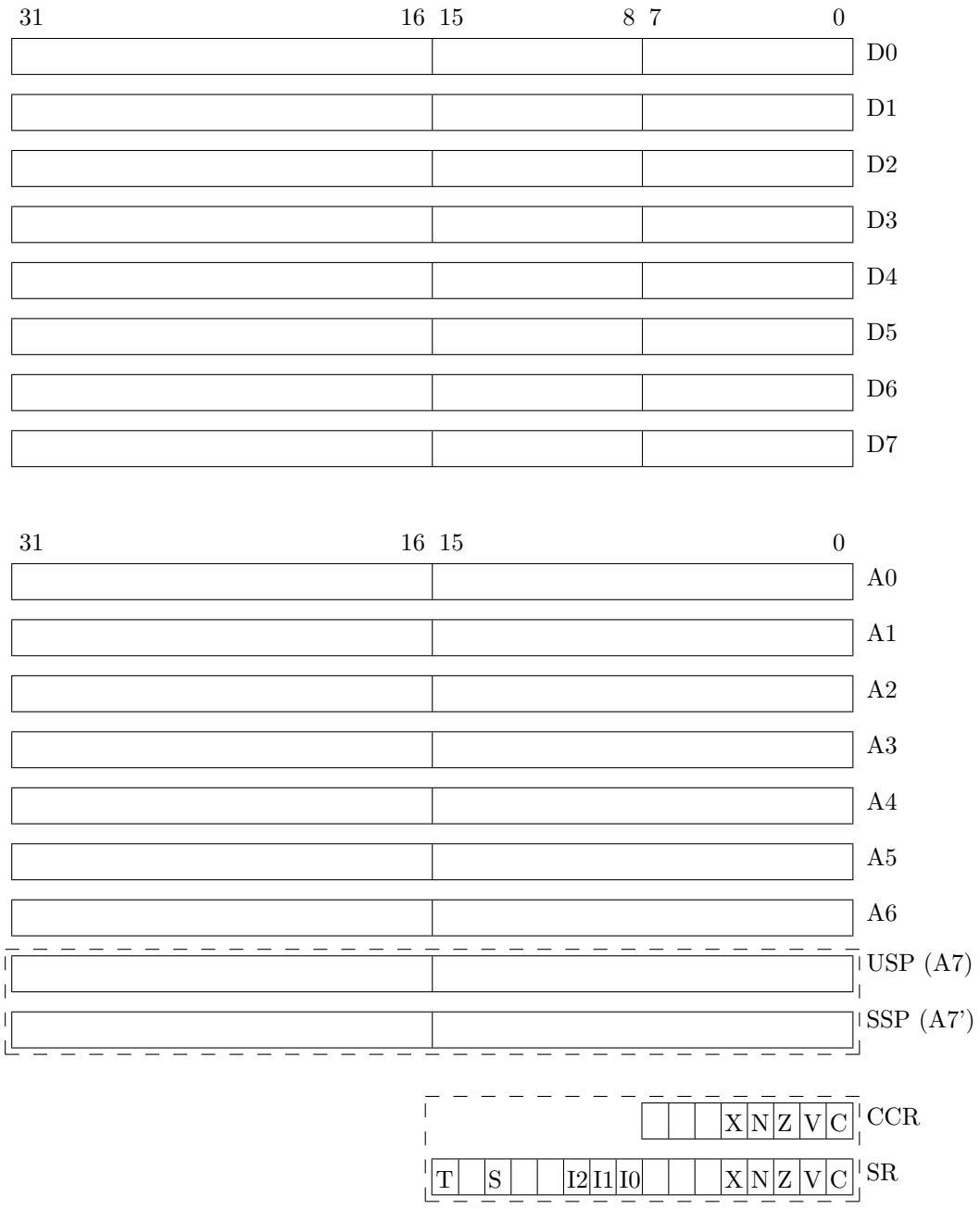
Układy wejścia/wyjścia komputerów są szczególnie ważne ze względu na konieczność współpracy z otoczeniem (w tym z człowiekiem). Obsługa urządzeń zewnętrznych odbywa się przy pomocy rozkazów specjalnych, lub przy pomocy takich samych rozkazów, jakimi można operować na komórkach pamięci. Wiąże się to bezpośrednio z dwoma podstawowymi koncepcjami umieszczenia układów we/wy.
Jedna z nich polega na wydzieleniu osobnej magistrali we/wy (*isolated i/o*). Procesor odwołuje się do urządzeń zewnętrznych przez specjalne rozkazy, które mogą zajmować nieco mniej miejsca i działać nieco szybciej, niż odwołania do danych w pamięci. Zazwyczaj ilość tych rozkazów jest ograniczona (w drastycznym przypadku do dwóch: *IN* i *OUT*). Dodajmy, że często oddzielenie magistrali we/wy polega na wydzieleniu tylko części sygnałów sterujących (w skrajnym przypadku – jedmego: *memory/io*) i wspólnym użytkowaniem linii adresowych i danych przes pamięć i układy we/wy.
Druga koncepcja polega na traktowaniu układów we/wy tak samo jak zwykłych komórek pamięci (*memory-mapped i/o*). Tu można używać wszystkich instrukcji do operowania na urządzeniach zewnętrznych, co z nawiązką rekompensuje straty spowodowane większą długością kodu pojedynczego rozkazu. Warto w tym miejscu zwrócić uwage, że nie zawsze da się skorzystać z wygodnych operacji przeznaczonych do działania na komórkach pamięci. Proszę rozważyć przykład jakiejkolwiek operacji modyfikującej słowo typu RMW (*read-modify-write*). Zadziała ona poprawnie tylko wtedy, gdy rejestr układu we/wy zachowuje się jak komórka pamięci, to znaczy pozwala odczytywać wpisaną doń wartość. W wielu przypadkach rzeczywistych tak nie jest. Dla oszczędności przestrzeni adresowej, pod tym samym adresem bywają umieszczane dwa różne rejestry dostępne w zależności od kierunku przesyłania danych. Może to być na przykład rejestr sterujący w czasie zapisu, a rejestr stanu przy odczycie. Wykonanie na takim rejestrze operacji np. negacji spowodowałoby zupełnie niezamierzony skutek: do rejestru sterującego zostałby wpisany zanegowany obraz rejestru stanu.

1.2 Organizacja M68000

Procesor MC68000, którym będziemy się zajmować w dalszym ciągu, jest maszyną o systemie we/wy adresowanym wspólnie z pamięcią. Wykorzystuje instrukcje jedno– i dwuadresowe. W tym (nominalnie szesnastobitowym) procesorze wszystkie wewnętrzne rejestry adresowe i danych są trzydziestodwubitowe. Pozwoliło to na rozwój rodziny aż do obecnego 68060 bez zmieniania garnituru rejestrów, tylko przez wprowadzanie drobnych rozszerzeń nie wpływających na zgodność z pierwszym modelem. Lista rozkazów tego procesora zawiera tylko 56 symbolicznych nazw instrukcji (*mnemonics*). Jej obszerność zawdzięczamy głównie dużej ilości trybów adresowania (16), które ułatwiają implementowanie różnych struktur danych i oprogramowywanie systemów operacyjnych. Ponieważ większość trybów adresowania może być zastosowana do większości instrukcji, oraz ze względu na symetryczny zestaw rejestrów, można mówić o **prawie** ortogonalnej strukturze listy rozkazów 68000.

1.2.1 Rejestry

Procesor 68000 odróżnia się wyraźnie od swoich poprzedników, jak również od współczesnych sobie rywali ilością uniwersalnych rejestrów i szeroką gamą dostępnych trybów adresowania, które z tych rejestrów korzystają. Wszystkie uniwersalne rejestry, zarówno danych (*Di*), jak i adresowe (*Aj*), mają długość 32 bitów. Podobnie ma się rzecz z licznikiem rozkazów (*PC*). Rejestr stanu procesora (*SR – Status Register*) jest 16-bitowy.
Licznik rozkazów (*PC – Program Counter*) pozwala na zaadresowanie 2^{32} bajtów pamięci, co daje 4GB przestrzeni adresowej. Ponieważ w pierwszym modelu rodziny M68K, procesorze MC68000, wyprowadzono na końcówki obudowy tylko 24 (a naprawdę 23, bez A0) linie adresowe, wyższe bity adresowe (A24 - A31) nie biorą udziału w dekodowaniu adresu na zewnątrz procesora, wobec czego adresowanie przebiega modulo 2^{24} i faktycznie dostępna przestrzeń adresowa (pamięci i urządzeń we/wy łącznie) maleje do 16MB. W dalszych modelach (od 68020) wszystkie linie adresowe są dostępne.
Rejestry danych (*D0 - D7*) mogą być używane jako 32–, 16– lub 8–bitowe. W przypadku wykorzystywania części rejestru, w operacjach bierze udział najmniej znacząca grupa bitów, bardziej znacząca część rejestru nie jest modyfikowana. Rejestry te pełnią rolę akumulatorów i biorą udział w niektórych trybach adresowania jako rejestry indeksowe.
Rejestry adresowe ogólnego przeznaczenia (*A0 - A6*) służą do przechowywania adresów komórek pamięci, które są szczególnie często wykorzystywane (tzw. adresy bazowe). Mogą to być adresy początków tablic, struktur



Rysunek 1: Rejestry procesora MC68000

itp. Dzięki ich wykorzystaniu w różnych trybach adresowania można zaoszczędzić wielokrotnego pobierania z pamięci pełnego adresu (w tym przypadku 4 bajty), poprzestając na znacznie krótszym adresie względnym wewnątrz struktury (tablicy). Daje to podwójną oszczędność: 4 bajty kodu i czas ich pobierania z pamięci przy dekodowaniu instrukcji.
Rejestr adresowy *A7* ma wyznaczoną dodatkową, ważną funkcję. Pełni rolę wskaźnika stosu dla wywołań podprogramów i obsługi zdarzeń specjalnych (*exceptions*). Przy wywołaniu podprogramu (*JSR - Jump to SubRoutine*) do komórek pamięci adresowanych przez zawartość *A7* składowana jest dotychczasowa zawartość licznika rozkazów (*PC*), stanowiąca adres, spod którego należałoby pobrać kod kolejnej instrukcji maszynowej po rozkazie *JSR*. Dzięki temu, przy wywołaniu rozkazu zakończenia poprogramu (*RTS - ReTurn from Subroutine*), stan licznika rozkazów może być odtworzony i sekwencyjne wykonanie programu podjęte na nowo.
Przy okazji trzeba dodać, że procesor 68000 może się znajdować w jednym z dwóch stanów:

- tryb pracy użytkownika (*user-mode*)
- tryb pracy systemowy (uprzywilejowany – *supervisor-mode*)

Rejestr *A7* występuje w dwóch wcieleniach: *USP – User Stack Pointer* i *SSP – Supervisor Stack Pointer*. W systemowym trybie pracy procesor może wykonywać wszystkie rozkazy i ma dostęp do wszystkich zasobów (w tym również do *USP*). W trybie pracy użytkownika niektóre instrukcje (w szczególności te, przy pomocy których możliwe byłoby przeprowadzenie procesora w systemowy tryb pracy) stają się nielegalne. Układ dekodowania operacji wykrywa to i generuje zdarzenie specjalne odpowiadające naruszeniu uprzywilejowania (*privillege violation*). Podobnie dzieje się przy próbie dostępu do chronionych zasobów (jak na przykład *SSP* w trybie użytkownika). Dzięki temu twórca oprogramowania systemowego może łatwo skonstruować zabezpieczenia przed błędami zwykłych użytkowników.
Rejestr stanu procesora (*SR*) ma również dwa wcielenia. W systemowym trybie pracy dostępny jest w całej (16-bitowej) okazałości. Zwykły użytkownik może operować tylko na jego mniej znaczącym bajcie zwanym rejestrem flagowym (*CCR – Condition Code Register*). Jeden z bitów niedostępnej części *SR* (bit 13, zwany *S*) decyduje o aktualnym trybie pracy procesora. Można więc porzucić tryb systemowy (przez modyfikację rejestru *SR*), lecz nie można w ten sam sposób doń wrócić. Ogólnie mówiąc, procesor wchodzi w systemowy tryb pracy **tylko** w wyniku zdarzeń specjalnych, do których należą przerwania, błędy wykrywane sprzętowo i restart procesora.

1.2.2 Organizacja pamięci

Jak już wspomniano przy omawianiu licznika rozkazów, najmniejszą jednostką pamięci jest ośmiobitowy bajt. Faktyczny rozmiar magistrali danych w procesorze 68000 wynosi 16 bitów, więc równocześnie można po niej przekazywać dwa bajty, wobec czego pamięć musi być dostępna przez 16-bitową bramę. W czasie jednego dostępu do pamięci (cyklu magistrali danych) wystawiany jest jeden adres na magistrali adresowej. Łatwo zauważyć, że adres A0 nie bierze udziału w wyborze 16-bitowego słowa na magistrali. W związku z tym dwa kolejne bajty w pamięci tworzą jedno słowo. Motorola przyjęła tzw. leksykalną konwencję kolejności bajtów w słowie (im starszy bajt, tym wcześniej występuje), odwrotną w stosunku do konwencji Intela, lecz równie rozpowszechnioną na świecie (proszę zwrócić uwagę, że jest ona naturalna dla ludzi piszących liczby od lewej ku prawej stronie, począwszy od najbardziej znaczącej cyfry). Z tej konwencji wynika, że słowa 16-bitowe (dla skrócenia zapisu będziemy je dalej nazywali po prostu słowami – *word*) są dostępne na magistrali danych pod warunkiem, że ich starszy bajt ma adres parzysty. O słowach takich mówimy, że są dopasowane (*aligned*). Nieparzyste (*misaligned*) słowa nie mogą być odczytane w jednym cyklu magistrali, trzeba wystawić dwa kolejne adresy. Procesor 68000 odmawia dostępu do takich słów i generuje zdarzenie specjalne (błąd adresowy – *address error*). Następne wersje zostały rozbudowane sprzętowo i pozwalają na nieparzystoadresowe słowa, lecz również zużywają więcej cykli na przesłanie słowa niezgodnego z ograniczeniami magistrali.
Urządzenia zewnętrzne, adresowane wspólnie z pamięcią, muszą być umieszczone w pewnym (wybranym przez projektanta) obszarze przestrzeni adresowej. Nie muszą one być dostępne na całej szerokości magistrali danych, wystarczy użyć połowy (1 bajt). Wprawdzie, zgodnie z wcześniejszymi rozważaniami, ich wewnętrzne rejestry nie będą zajmowały kolejnych adresów, ale i tak zwykle nie ma to istotnego znaczenia. W ten sposób można z procesorami M68000 używać portów z wcześniejszych, 8-bitowych rodzin mikroprocesorowych (szczególnie łatwo z rodziny 6800). Więcej informacji o obsłudze wejść/wyjść (szczególnie o przerwaniach) podamy w dalszym ciągu wykładu.

1.3 Podstawowe typy danych w M68000

Typy danych stosowane w językach wyższego rzędu, jak Pascal lub C, można podzielić na podstawowe (proste) i złożone. Pierwsze z nich mają bezpośrednią implementację w języku maszynowym, drugie (jak się później przekonamy) czasem też, choć na ogół wymagają specjalnych technik dla uzyskania efektywnej implementacji, wykorzystujących podstawowe typy danych jako elementy składowe.
Procesory M68K mają następujące podstawowe typy danych:

- bit,
- cyfra dziesiętna (BCD),
- bajt,
- słowo (16-bitowe),
- długie słowo (32-bitowe).

Każdy z powyższych typów danych posiada własne odwzorowanie w pamięci, która, jak pamiętamy, ma organizację bajtową.

1.3.1 Bit

Bit może mieć jedą z dwóch wartości: 1 lub 0. Może być umieszczony na dowolnej z ośmiu pozycji w bajcie (komórce pamięci) lub na dowolnej z 32 pozycji w rejestrze danych procesora. Adres względny bitu mieści się wtedy (odpowiednio) w przedziale 0-7 lub 0-31.

1.3.2 Cyfra dziesiętna (BCD)

Binarna reprezentacja cyfr dziesiętnych (*BCD – Binary Coded Decimal*) bywa używana (raczej rzadko) przy niektórych zastosowaniach mikroprocesorów. Polega ona na zapisie pozycyjnym liczb przy podstawie dziesiętnej z równoczesnym naturalnym (binarnym) kodowaniem cyfr dziesiętnych. Nie jest to najefektywniejszy sposób kodowania liczb, ponieważ na 16-bitowym słowie można zmieścić tylko liczby BCD o zakresie 0 – 9999, podczas gdy binarny kod pozwala zmieścić liczby od 0 do 65535. Każda cyfra w tym kodzie zajmuje cztery bity. Mamy do dyspozycji dwa warianty liczb BCD: upakowane i nieupakowane. Pierwszy z nich mieści w każdym bajcie pamięci dwie cyfry dziesiętne, przy czym starsza z nich zajmuje starszą część bajtu, drugi wykorzystuje tylko mniej znaczącą połowę bajtu na jedną cyfrę.

1.3.3 Bajt

Bajt jest podstawowym elementem, z którego zbudowana jest pamięć procesorów 68000. Może on być wykorzystany do przechowywania znaków (*ASCII – American Standard Code for Information Interchange*), lub małych liczb binarnych (w zakresie od 0 do $2^8 - 1 = 255$) zakodowanych naturalnie. Można też w bajcie przechowywać liczby ze znakiem. Motorola używa reprezentacji liczb ze znakiem (*signed byte*) w kodzie uzupełnienia do 2 (*two's complement*). Daje to zakres od $-2^7 = -128$ do $+2^7 - 1 = +127$.

1.3.4 Słowo

Domyślnym rozmiarem słowa w procesorach 68000 są dwa bajty (16 bitów). Daje to zakres liczb całkowitych bez znaku (*unsigned integers*) od 0 do $2^{16} - 1 = 65535$, oraz ze znakiem od $-2^{15} = -32788$ do $+2^{15} - 1 = +32787$. Zgodnie z wcześniejszymi rozważaniami dotyczącymi pamięci, słowa muszą być umieszczone począwszy od parzystych adresów, starszy bajt słowa pod adresem parzystym, młodszy pod następnym (nieparzystym).

1.3.5 Długie słowo

Dwa słowa 16-bitowe tworzyć mogą długie słowo (*longword*). Zakres liczb całkowitych bez znaku wynosi dla długiego słowa od 0 do $2^{32} - 1 = 4294963295$, a ze znakiem od $-2^{31} = -2147481648$ do $+2^{31} - 1 = 2147481647$. Tu również obowiązują te same uwagi dotyczące kolejności bajtów i umieszczenia począwszy od adresów parzystych.

1.3.6 Reprezentacja instrukcji

Kod operacji może zajmować w słowie rozkazowym pole o stałej ilości bitów (wielkości). Na przykład 8-bitowe pole kodu operacji dawałoby 256 możliwych kodów. Ponieważ jednak nie każdy rozkaz wymaga takiej samej ilości argumentów, a co za tym idzie - dodatkowych bitów w słowie rozkazowym, w procesorach Motoroli rodziny M68K zastosowano zmiennej długości pole kodu operacji. Rozpiętość tego pola wynosi od 2 do 16 bitów.
Kody operacji (*opcodes*) można podzielić na:

- bezadresowe (*implicit address opcodes*)
- jednoadresowe (*single-address opcodes*)
- półtoraadresowe (*one-and-a-half-address opcodes*)
- dwuadresowe (*two-address opcodes*)

Odpowiednio dla każdej z tych kategorii możemy podać reprezentantów:

- `RTS` (ReTurn from Subroutine) - powrót z przerwania
- `JMP <ea>` (JuMP) - skok do adresu
- `ADD <ea>,Dn` - dodanie zawartości adresu do Dn
- `MOVE <ea>,<ea>` - przesłanie zawartości jednego adresu pod drugi

Operacje bezadresowe nie używają informacji adresowej. Rozkazy takie mają kody 16-bitowe .Przykładowy rozkaz `RTS` ładuje zawartość licznika rozkazów (*PC*) z komórki pamięci wskazywanej przez wskaźnik stosu (*SP*):

```
RTS                         ((SP)) -> PC
```

Operacje jednoadresowe wykorzystują 6-bitowe pole w słowie rozkazowym zwane deskryptorem adresu (*effective address*), które jest w różny sposób wykorzystywane do tworzenia wynikowego adresu 32-bitowego. Sposoby te, zwane trybami adresowania zostaną omówione w kolejnym rozdziale. Rozkaz `JMP <ea>` ładuje wynikowy adres otrzymany z `<ea>` do licznika rozkazów:

```
JMP     <ea>                <ea> -> PC
```

Przykładem następnej klasy jest `ADD`. Ta operacja używa dwóch argumentów. Jeden z nich może być dostępny przez `<ea>` analogicznie, jak dla operacji jednoadresowych, a drugi **musi** być w rejestrze danych:

```
ADD     <ea>,Dn             (<ea>) + (Dn) -> Dn
ADD     Dn,<ea>             (Dn) + (<ea>) -> <ea>
```

Operacją dwuadresową jest `MOVE`. Przesyła ona dane z jednej do drugiej lokacji:

```
MOVE    <eas>,<ead>         (<eas>) -> <ead>
```

Operacje mogą specyfikować rozmiar danych, na których operują. Dostępne (nie we wszystkich przypadkach) rozmiary to bajt, słowo i długie słowo.
Reasumując, słowo rozkazowe zawiera kod operacji, specyfikację rozmiaru danych, i od zera do dwóch deskryptorów adresu:

```
RTS                     0       % 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 1
JMP     <ea>            1       % 0 1 0 0 1 1 1 0 1 1 <---ea---->
ADD.B   Dn,<ea>         1.5     % 1 1 0 1 <-Dn> 1 0 0 <---ea---->
ADD.W   Dn,<ea>         1.5     % 1 1 0 1 <-Dn> 1 0 1 <---ea---->
ADD.L   Dn,<ea>         1.5     % 1 1 0 1 <-Dn> 1 1 0 <---ea---->
MOVE.B  <eas>,<ead>     2       % 0 0 0 1 <---ead---> <---eas--->
MOVE.W  <eas>,<ead>     2       % 0 0 1 1 <---ead---> <---eas--->
MOVE.L  <eas>,<ead>     2       % 0 0 1 0 <---ead---> <---eas--->
```

Programowanie w języku asemblera nie wymaga, na szczęście, znajomości binarnych kodów operacji. Przytoczone przykłady mają tylko pomóc w uzmysłowieniu pewnych faktów, które pojawią się przy omawianiu trybów adresowania.

1.4 Adresowanie

Podstawowym problemem przy dostępie do danych zawartych w pamięci jest sposób podania adresu, pod który chcemy się zwrócić. Metoda bezpośrednia, choć oczywista, ma jednak podstawową wadę, którą jest konieczność umieszczenia w kodzie maszynowym programu pełnych adresów. Mogą one mieć długość 32 bitów (jak to jest w omawianych procesorach) i przez to zajmować po cztery bajty na adres. Rejestry dostępne w jednostce centralnej pozwalają złagodzić ten problem przez tymczasowe przechowywanie najczęściej potrzebnych danych. Inną metodą jest użycie różnych sposobów adresowania, dopasowanych do specyficznych potrzeb. Duża ilość trybów adresowania w procesorach rodziny M68K Motoroli usprawnia pracę programów, ale równocześnie wymaga od programisty ich zrozumienia i opanowania technik ich stosowania.
Przedstawimy tryby adresowania od najprostszych do najbardziej złożonych, posługując się w miarę potrzeby odwołaniami do analogii z języka C (którego znajomość założyliśmy na wstępie).
Nasze rozważania dotyczące trybów adresowania będziemy ilustrować przykładami. Weźmy instrukcję:

```
MOVE.W  <skad>,<dokad>
```

przesyłającą słowo dwubajtowe (`.W`) z miejsca określonego specyfikacją `skad` do miejsca określonego przez `dokad`. Zawartość źródła nie ulega zmianie, w miejscu przeznaczenia dwubajtowe słowo zostaje zastąpione wartością taką, jaką ma źródło, a rejestr flagowy (*CCR*) zostanie odpowiednio zmodyfikowany.
Instrukcja `MOVE.W` jest zakodowana następująco:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   0   1   1  '----------v----------' '----------v----------'
          kod    rozmiar   adres przeznaczenia        adres zrodla
        operacji danych   (dokad - destination)      (skad - source)
```

Kod operacji `MOVE` zajmuje tylko dwa bity (00), rozmiar danych jest też dwubitowy (01 - bajt, 11 - slowo, 10 - długie słowo). Adresy źródła i przeznaczenia, są określone na sześciobitowych polach w słowie zawierającym kod maszynowy instrukcji `MOVE.W`. Pole to składa się z dwóch trzybitowych części. Jedna z nich, zwana jest trybem (*Mode*), a druga stanowi numer rejestru (*Register*). Nie wszystkie tryby adresowania używają rejestrów. Dla nich zarezerwowano wspólną wartość pola trybu (`%111`), przy której pole rejestru stanowi właściwy wyróżnik trybu adresowania. W ten sposób jednostka centralna po pobraniu pierwszego słowa rozkazu może ustalić jednoznacznie tryby, przy pomocy których zaadresowano argumenty operacji. Oczywiście, większość trybów będzie wymagała pobrania dalszych słów składających się na pełny rozkaz.

1.4.1 Tryb bezpośredni rejestru danych (*data register direct*)

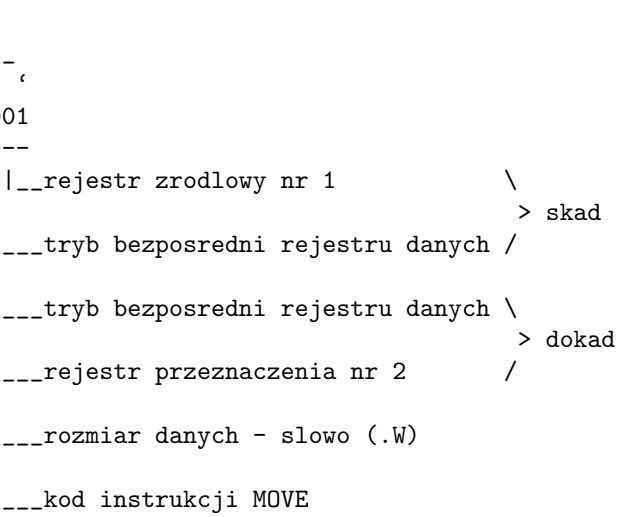
Tryb ten umożliwia dostęp do danych zawartych w rejestrach danych procesora. Pole trybu ma wartość `%000`, a w polu rejestru podany jest numer wybranego rejestru danych. Adres efektywny odnosi się bezpośrednio do rejestru danych (nie ma dostępu do pamięci):

$$EA = Dn$$

W naszym przykładzie:

```
        MOVE.W  D1,D2                       (D1) -> D2
```

koduje się:



Jak widać, ta instrukcja zajmuje tylko jedno słowo, co daje jednoczesną oszczędność miejsca w pamięci programu i czasu zużytego na jej wykonanie (tylko jeden dostęp do pamięci).

1.4.2 Tryb bezpośredni rejestru adresowego (*address register direct*)

Tryb ten jest podobny do porzedniego (pole trybu `%001`), ale może być stosowany tylko dla danych o długości słowa (`.W`) lub długiego słowa (`.L`). Adres efektywny odnosi się bezpośrednio do rejestru adresowego (nie ma dostępu do pamięci):

$$EA = An$$

Nasz przykład:

```
        MOVE.W  D1,A2                       (D1) -> A2
```

koduje się:

```
 00 11 010 001 000 001
 -- -- --- --- --- ---
  |  |  |   |   |   |__rejestr zrodlowy nr 1               \
  |  |  |   |   |                                          > skad
  |  |  |   |   |______tryb bezposredni rejestru danych    /
  |  |  |   |
  |  |  |   |___________tryb bezposredni rejestru adresowego \
  |  |  |                                                   > dokad
  |  |  |_______________rejestr przeznaczenia nr 2          /
  |  |
  |  |__________________rozmiar danych - slowo (.W)
  |
  |______________________kod instrukcji MOVE
```

Również w tym trybie oszczędzamy zarazem na długości kodu, jak i na czasie jego wykonania. Obu omówionych trybów używa się do operowania na często wykorzystywanych danych, pośrednich wynikach itp.

1.4.3 Tryb bezwzględny długi (*absolute long*)

Operowanie na danych (nawet w przypadku omówionych trybów rejestrowych) wymaga kontaktu z zawartością pamięci (choćby po to, by załadować dane do rejestrów i przekazać wyniki). Omawiany tryb pozwala zaadresować dowolną komórkę pamięci w całej (4GB) przestrzeni adresowej procesora. Wymaga to podania czterobajtowego (dwa słowa) adresu. W tym trybie nie bierze udziału żaden rejestr, więc specyfikacja trybu ma pole `%111`, a pole rejestru zawiera `%001`. Ten (w sumie sześciobitowy) kod trybu adresowania powoduje, że procesor pobiera dwa słowa następujące po kodzie maszynowym rozkazu i składa je w długie słowo traktowane jako adres komórki pamięci, do której ma się odwołać:

$$EA = ((PC) + 2) : ((PC) + 4)$$

Asembler (program tłumaczący) na podstawie tekstu źródłowego:

```
DANEWEJ     EQU     $123456

            MOVE.W  DANEWEJ,D2                  (DANEWEJ) -> D2
```

gdzie `EQU` jest pseudoinstrukcją (dyrektywą) przypisującą etykiecie `DANEWEJ` wartość `$123456`, wytworzy kod maszynowy:

```
    $3435               kod maszynowy MOVE.W z odpowiednimi trybami
    $0012               starsze slowo dlugiego adresu bezwzglednego
    $3456               mlodsze slowo dlugiego adresu bezwzglednego
```

Jego pobranie wymaga trzech odwołań do pamięci, a wykonanie - kolejnego dostępu w celu odczytania słowa `DANEWEJ` zawartego w komórce o adresie `$123456`. W przypadku instrukcji:

```
DANEWEJ     EQU     $123456
DANEWYJ     EQU     $345678

            MOVE.W  DANEWEJ,DANEWYJ         (DANEWEJ) -> DANEWYJ
```

pobranie rozkazu zajmie pięć dostępów, a wykonanie - szósty. Widać, że ten tryb adresowania powinien być stosowany ze szczególnym umiarem. Jego cechą jest (zgodnie z nazwą) **bezwzględna** specyfikacja adresu. W pewnych przypadkach jest to zaleta, gdyż pozwala odwoływać się do ustalonych położeń w przestrzeni adresowej procesora (np. komórek związanych z rezydentnym oprogramowaniem, portów). Z drugiej strony, nie jest to tryb godny polecenia w celu odwoływania się do roboczych danych programu, o ile chcielibyśmy uzyskać kod niezależny od położenia w pamięci (*PIC - Position Independent Code*), a tym bardziej kod współużywalny (*sharable/reentrant code*).

1.4.4 Tryb bezwzględny krótki (*absolute short*)

Nie zawsze istnieje potrzeba stosowania 32-bitowego adresu komórki pamięci. Wydzielono podobszar przestrzeni adresowej, który można adresować przy pomocy adresu 16-bitowego. Tryb bezwzględny krótki ma pole trybu `%111` i pole rejestru `%000`. Procesor obsługuje go w ten sposób, że pobiera następne słowo po kodzie rozkazu i traktuje je jako liczbę zapisaną w kodzie uzupełnienia do 2 (*U2*). Aby uzyskać 32-bitowy adres, przeprowadza operację rozciągnięcia znaku (*sign extension*), która polega na uzupełnieniu brakujących, bardziej znaczących bitów (w tym przypadku całego starszego słowa) zgodnie z najstarszym (tu – 15) bitem słowa rozszerzanego. Dzięki temu uzyskuje liczbę ze znakiem (*U2*) o tej samej wartości, lecz większej długości. W rezultacie otrzymuje adres, który dotyczy albo najniższych, albo najwyższych 32kB przestrzeni adresowej procesora:

$$EA = ((PC) + 2)_{SEX}$$

Zwróćmy uwagę, że najwyższa część przestrzeni adresowej jest rozumiana z dokładnością do ilości bitów adresowych wykorzystywanych do sprzętowego dekodowania adresu pamięci i urządzeń. Na przykład, używając tylko adresów do A19 włącznie, uzyskamy w tym trybie dostęp do obszarów:

```
   $00000 --    $07FFF   i      $F8000 --    $FFFFF
```

choć wewnętrzna reprezentacja adresów będzie nadal 32-bitowa:

```
$00000000 -- $00007FFF   i   $FFFF8000 -- $FFFFFFFF
```

Asembler w czasie tłumaczenia programu preferuje tryb bezwzględny krótki. Warunkiem jego zastosowania jest znajomość liczbowej wartości adresu odpowiadającego ewentualnej nazwie symbolicznej (etykiecie) **w chwili tworzenia kodu rozkazu**. W przypadku niespełnienia tego warunku wybierany jest tryb bezwzględny długi, jako zawsze realizowalny, choć czasem nieoptymalny. W większości asemblerów istnieje możliwość wymuszenia trybu długiego przez opatrzenie stałej liczbowej przyrostkiem `.L` lub użycie zer nieznaczących.

1.4.5 Tryb natychmiastowy (*immediate*)

W opisanych trybach bezwzględnych dane pobrać można z komórki pamięci, której adres podamy w kodzie programu. Często istnieje potrzeba użycia danych stałych, które mogą stanowić na przykład granice pętli, współczynniki wyrażeń itp. Ich umieszczenie w komórkach pamięci jest oczywiście możliwe, ale nieoptymalne. Zamiast pobierać z kodu programu słowa składające się na adres komórki zawierającej stałą wartość i następnie pobierać tę wartość z pamięci danych, można ją pobrać bezpośrednio z kodu programu. Oszczędzamy w ten sposób zarówno komórkę pamięci przeznaczoną na stałą, jak i czas potrzebny na dostęp do niej.
Tryb natychmiastowy w procesorach M68K jest symbolicznie wskazywany w tekście źródłowym programu przez poprzedzenie argumentu znakiem "`#`":

```
STALA       EQU     $55AA

            MOVE.W  #STALA,D2               STALA -> D2
```

Pole trybu wynosi `%111`, a pole rejestru `%100`. Rozmiar użytych danych zależy od specyfikacji w kodzie operacji:

- `.W` – słowo 16-bitowe, $EA = (PC) + 2$
- `.L` – długie słowo (32-bitowe), $EA = (PC) + 2$
- `.B` – 8-bitowy bajt, $EA = (PC) + 3$

Brak specyfikacji rozmiaru oznacza domyślnie słowo 16-bitowe. Zarówno słowo, jak i bajt zajmują w kodzie maszynowym drugie słowo po kodzie rozkazu, natomiast stała o rozmiarze długiego słowa - dwa kolejne słowa (podobnie jak w przypadku długiego adresu).
Zwróćmy uwagę, że stałe w programie nie powinny być kodowane w postaci "magicznych liczb". Używanie ich w postaci symbolicznej po nadaniu wartości dyrektywą `EQU` pozwala uporządkować program i procentuje przy konieczności zmiany stałych wartości przy jego modyfikowaniu.
Dążenie do skracania kodu maszynowego programów przejawia się we wprowadzeniu krótkiej odmiany danych natychmiastowych (*quick immediate constants*). Dla instrukcji `MOVE` istnieje odmiana `MOVEQ`, która korzysta z 8-bitowego pola wygospodarowanego w słowie zawierającym kod rozkazu, podobne pola 3-bitowe wygospodarowano w instrukcjach `ADDQ` i `SUBQ` będących mutacjami dodawania i odejmowania, a właściwie – uogólnieniem operacji inkrementacji i dekrementacji rejestrów danych.

1.4.6 Tryb pośredni rejestru adresowego (*address register indirect*)

Wyobraźmy sobie, że w naszym systemie musimy często dostawać się do portu (rejestru urządzenia zewnętrznego), którego adres jest długi. Z dotychczas prezentowanych trybów nadawał się do tego celu bezwzględny długi. Rozkaz wpisania 16-bitowych danych natychmiastowych do tego portu:

```
Dane        EQU     $55AA               #define Dane 0x55aa
PortAdd     EQU     $FFF000             #define PortAdd 0xfff000

* Deklaracja zmiennych
            ORG     PortAdd
Port        DS.W    1                   short Port @ Portadd;

* Przeslanie danych do portu
            MOVE.W  #Dane,Port          Port = Dane;
```

zajmuje przy każdym dostępie do portu 4 słowa 16-bitowe, a jego pobranie i wykonanie wymaga 5 dostępów do pamięci.
Znacznie lepszym sposobem jest ustawienie w jednym z rejestrów adresowych procesora wskaźnika na ten port i wykorzystanie go do pośredniego adresowania:

```
Dane    EQU     $55AA               #define Dane 0x55aa
PortAdd EQU     $FFF000             #define PortAdd 0xfff000

* Deklaracje zmiennych
*                                   short *ptr; /* rejestr A0 */

* Inicjacja wskaznika
            ORG     ROM
            MOVEA.L #PortAdd,A0     ptr = (short *)PortAdd;

* Przeslanie danych do portu
            MOVE.W  #Dane,(A0)      *ptr = Dane;
```

Notacja źródłowa `(A0)` oznacza, że zawartość rejestru `A0` jest adresem, do którego chcemy sięgnąć:

$$EA = (An)$$

Pole trybu wynosi `%010`, wskaźnikiem może być tylko rejestr adresowy.
Teraz wprawdzie jednorazowa inicjacja wskaźnika w rejestrze A0 zajmuje 3 słowa (3 dostępy), ale za to rozkaz wpisu danych natychmiastowych do portu skraca się do 2 słów (3 dostępy). Łatwo sprawdzić, że opłaca się to już przy dwukrotnym dostępie do portu przy jednej inicjacji wskaźnika.

1.4.7 Tryb pośredni rejestru adresowego z autopostinkrementacja (*address register indirect with postincrement*)

Rozszerzona wersja pośredniego trybu adresowania pozwala uzyskać sekwencyjny dostęp do kolejnych komórek pamięci bez wydłużania kodu maszynowego programu i czasu jego realizacji. Znana np. z języka C operacja autoinkrementacji wskaźnika po wykonaniu dostępu do danych została zaimplementowana bezpośrednio w jednym z trybów adresowania procesorów M68K. Dzięki temu przepisywanie zawartości jednego obszaru pamięci do drugiego nie wymaga dodatkowych operacji na wskaźnikach:

```
* Deklaracje zmiennych
            ORG     RAM
*                                   short *ptrA; /* rejestr A0 */
*                                   short *ptrB; /* rejestr A1 */
TablA       DS.W    100             short TablA[100];
TablB       DS.W    100             short TablB[100];

* Inicjacja wskaznikow
            ORG     ROM
            MOVEA.L #TablA,A0       ptrA = &TablA[0];
            MOVEA.L #TablB,A1       ptrB = &TablB[0];

* Przepisanie elementow i przesuniecie wskaznikow
            MOVE.W  (A0)+,(A1)+     *ptrB++ = *ptrA++;
```

Pole trybu tej operacji wynosi `%011`, jako wskaźniki mogą być używane tylko rejestry adresowe. Po wystawieniu adresu na magistralę, w celu uzyskania dostępu do wybranej komórki pamięci, zawartość rejestru adresowego jest zwiększana o rozmiar danych ($SIZE$) biorących udział w operacji wyrażony w bajtach (”`.B`” – 1, ”`.W`” – 2, ”`.L` – 4):

$$EA = (An)$$
$$(An) + SIZE \rightarrow An$$

Dzięki temu wskaźnik jest ustawiony na następny element tablicy, a ponieważ operacja ta odbywa się w rejestrze wewnętrznym procesora, jej wykonanie nie zajmuje dodatkowego czasu.

1.4.8 Tryb pośredni rejestru adresowego z autopredekrementacja (*address register indirect with predecrement*)

W niektórych sytuacjach potrzebne jest przepisywanie tablic w odwrotnej kolejności, na przykład przy częściowym pokrywaniu się obszaru źródłowego z docelowym. Wtedy można użyć trybu przeciwstawnego do poprzednio opisanego :

```
* Przesuniecie wskaznikow i przepisanie elementow
            MOVE.W  -(A0),-(A1)     *--ptrB = *--ptrA;
```

Kod trybu wynosi `%100`, rejestr musi być adresowy. Tym razem modyfikacja (odpowiednie do typu danych zmniejszenie zawartości) rejestru adresowego następuje **przed** wystawieniem adresu na magistralę:

$$(An) - SIZE \rightarrow An$$
$$EA = (An)$$

Ważnym zastosowaniem trybów pośrednich z automodyfikacją są operacje na stosie. Istnieje wersja instrukcji `MOVE`, zwana `MOVEM` (*MOVE Multiple*), która pozwala przesłać na stos lub pobrać z niego dowolny podzbiór rejestrów danych i adresowych procesora przy pomocy rozkazu złożonego z dwóch słów: kodu rozkazu i maski zestawu rejestrów.

1.4.9 Tryb pośredni rejestru adresowego z przesunięciem (*address register indirect with displacement*)

Innym rozszerzeniem pośredniego trybu rejestru adresowego jest dodanie stałego przesunięcia (przemieszczenia) adresu wynikowego w stosunku do zawartości rejestru adresowego, zwanej adresem bazowym. Przesunięcie to jest podawane w postaci 16-bitowego słowa po kodzie rozkazu, traktowanego jako liczba ze znakiem. Adres wystawiany na magistralę budowany jest z zawartości rejestru adresowego przez dodanie przesunięcia po rozciągnięciu jego znaku na słowo 32-bitowe:

$$EA = (An) + ((PC) + 2)_{SEX}$$

Typowe zastosowanie tego trybu to dostęp do danych wewnątrz struktur:

```
* Definicja struktury data_t
            ORG     0
                                    typedef struct{
Dzien       DS.B    1                   char Dzien;
Miesiac     DS.B    1                   char Miesiac;
Rok         DS.W    1                   short Rok;
*                                   } data_t;
Data_L
*                                   Data_L = sizeof(data_t);

* Deklaracje zmiennych
            ORG     RAM
*                                   data_t *ptr; /* w rejestrze A0 */
Data        DS.B    Data_L          data_t Data;

* Dostep do wnetrza struktury
            ORG     ROM
            MOVEA.L #Data,A0        ptr = &Data;
            MOVE.W  #1996,Rok(A0)   ptr->Rok = 1996;
```

Pole trybu wynosi `%101`, rejestr – adresowy, przesunięcie – 16-bitowe.

1.4.10 Tryb pośredni rejestru adresowego z indeksem i przesunięciem (*address register indirect with index and displacement*)

Dostęp do elementów tablic wymaga znajomości dwóch parametrów: adresu początku tablicy i kolejnego numeru (indeksu) elementu, o który nam chodzi. Oba te parametry możemy zadać w opisywanym trybie w postaci zawartości dwóch rejestrów. Pierwszy z nich **musi** być rejestrem adresowym i zawiera adres bazowy, a drugi może być dowolnym z uniwersalnych rejestrów procesora i jest zwany rejestrem indeksowym. Dodatkowa możliwość użycia stałego przesunięcia względem rejestru bazowego (jak poprzednio) pozwala na dostęp do tablic zawartych w strukturach. Adres efektywny tworzony jest przez zsumowanie zawartości rejestru bazowego, 8-bitowego przesunięcia względem bazy rozszerzonego znakowo i zawartości rejestru indeksowego (rozszerzonego w przypadku użycia jego 16-bitowej części):

$$EA = (An) + ((PC) + 3)_{SEX} + (Xm)_{SEX}$$

Oto ilustracja użycia opisywanego trybu w celu dostępu do tablicy pomiarów zawartej w pewnej strukturze:

```
* Definicja struktury pomiar_t
            ORG     0
                                    typedef struct{
Datas       DS.B    Data_L              data_t Datas;
Temp        DS.B    24                  char Temp[24];
*                                   } pomiar_t;
Pomiar_L
*                                   Pomiar_L = sizeof(pomiar_t);

* Deklaracje zmiennych
            ORG     RAM
*                                   pomiar_t *ptr; /* w rejestrze A0 */
*                                   int NumerPom;  /* w rejestrze D1 */
*                                   char Wynik;    /* w rejestrze D0 */
Pomiar      DS.B    Pomiar_L        pomiar_t Pomiar;

* Dostep do elementu tablicy wewnatrz struktury
            ORG     ROM

* Inicjacja wskaznika struktury
            MOVEA.L #Pomiar,A0          ptr = &Pomiar;

* Inicjacja indeksu w tablicy
            MOVE.L  #0,D1               NumerPom = 0;

* odczyt pomiaru z tablicy
            MOVE.B  Temp(A0,D1.L),D0    Wynik = ptr->Temp[NumerPom];
```

Przyjrzymy się teraz sposobowi kodowania trybu indeksowego w procesorze M68K. Pole trybu wynosi `%110`, a pole rejestru zawiera numer rejestru bazowego. W kolejnym słowie umieszczone są parametry dotyczące rejestru indeksowego i 8-bitowe przesunięcie bazowe:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   0   0   1   1   0   0   0   0   0   0   0   0   1   0   0
        D/A   '---v---' W/L               '-------------v-------------'
        typ /   numer /czesc                    przesuniecie bazowe
        rejestru indeksowego
```

Zwróćmy uwagę, że trybu tego można również użyć do szybkiej implementacji tablic dwuwymiarowych. Jeden z indeksów tablicy stanowić może rejestr bazowy, a drugi - rejestr indeksowy.

W procesorach M68K począwszy od MC68020 wprowadzono dodatkowo skalowanie indeksu, które polega na wykorzystaniu bitów b_9 i b_{10} drugiego słowa rozkazowego do zakodowania rozmiaru elementu tablicy:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   0   0   1   1   1   0   0   0   0   0   0   0   1   0   0
        D/A   '---v---' W/L   '-v-'       '-------------v-------------'
        typ /   numer /czesc  skala            przesuniecie bazowe
        rejestru indeksowego  elem.
```

Kodowanie jest naturalne, rozmiar elementu określony jest jako 2^{SKALA}. W poniższym przykładzie zmieniono typ elementów tablicy na `long`, by zlustrować użycie skalowania:

```
* Definicja struktury pomiar_t


            ORG     0
                                    typedef struct{
Datas       DS.B    Data_L              data_t Datas;
Temp        DS.L    24                  long Temp[24];
*                                   } pomiar_t;
Pomiar_L
*                                   Pomiar_L = sizeof(pomiar_t);

* Deklaracje zmiennych
            ORG     RAM
*                                   pomiar_t *ptr; /* w rejestrze A0 */
*                                   int NumerPom;  /* w rejestrze D1 */
*                                   long Wynik;    /* w rejestrze D0 */
Pomiar      DS.B    Pomiar_L        pomiar_t Pomiar;

* Dostep do elementu tablicy wewnatrz struktury
            ORG     ROM

* Inicjacja wskaznika struktury
            MOVEA.L #Pomiar,A0          ptr = &Pomiar;

* Inicjacja indeksu w tablicy
            MOVE.L  #0,D1               NumerPom = 0;

* odczyt pomiaru z tablicy
            MOVE.L  Temp(A0,D1.L*4),D0  Wynik = ptr->Temp[NumerPom];
```

1.4.11 Tryb pośredni licznika rozkazów z przesunięciem (*program counter indirect with displacement*)

Szczególne zalety ma możliwość użycia licznika rozkazów jako rejestru bazowego. Taki tryb adresowania pozwala stworzyć kod maszynowy działający niezależnie od położenia w pamięci (*PIC – Position Independent Code*). Adres jest tworzony przez dodanie rozszerzonego znakowo przesunięcia do zawartości licznika rozkazów **po** pobraniu pierwszego słowa rozkazu:

$$EA = (PC) + 2 + ((PC) + 2)_{SEX}$$

Pozornie skomplikowane zadanie obliczania wartości przesunięcia jest przerzucone na asembler, który oblicza różnicę pomiędzy bieżącą wartością swojego licznika adresowego, a wartością przypisaną nazwie (etykiecie) określającej miejsce w pamięci, do którego się odwołujemy.

```
            ORG     ROM
Napis       DC.B    'Ala ma kota'
            DC.B    0
*
* Dostep do pierwszego znaku napisu
            MOVE.B  Napis(PC),D0
```

Zalety tego trybu adresowania są jeszcze lepiej widoczne przy użyciu instrukcji `LEA` (*Load Effective Address*), która obliczony dla użytego w niej trybu adresowania adres wynikowy przesyła do zadanego rejestru adresowego. Dalej można tego rejestru użyć jako bazowego dla innych (w szczególności automodyfikowalnych lub indeksowych) trybów:

```
            ORG     ROM
Napis       DC.B    'Ala ma kota'
            DC.B    0
*
* Inicjacja rejestru bazowego
            LEA.L   Napis(PC),A0
*
* Dost/ep do kolejnych znakow napisu
            MOVE.B  (A0)+,Port
```

Pole trybu – `%111`, pole rejestru – `%010`, przesunięcie – 16-bitowe.
Zwróćmy uwagę, że prawidłowe zastosowanie tego trybu pozwala uzyskać niezależność kodu od położenia, jak również jego współużywalność (*sharable/reentrant code*). Opisany tryb jest przeznaczony do odczytu, a nie do zapisu danych. Nie stanowi to jednak żadnego ograniczenia. W systemach wielozadaniowych różne procesy mogą korzystać z tej samej kopii kodu w różnych chwilach. Nie mogą jednak modyfikować stanu zmiennych globalnych związanych z tym kodem. Dla zmiennych lokalnych stosuje się różne rozwiązania (lokalne wskaźniki obszaru danych statycznych dla procesu, dane dynamiczne – na stosie itp.).

1.4.12 Tryb pośredni licznika rozkazów z indeksem i przesunięciem (*program counter indirect with index and displacement*)

Analogicznie do poprzedniego, ten tryb pozwala się odwoływać do danych zawartych w kodzie programu. Tym razem moą to być tablice, jak to było przy trybie indeksowym z rejestrem bazowym. Pole trybu – `%111`, pole rejestru – `%011`. Teraz rolę rejestru bazowego pełni licznik rozkazów (jego zawartość jest zmodyfikowana o 2 po pobraniu kodu rozkazu):

$$EA = (PC) + 2 + ((PC) + 3)_{SEX} + (Xm)_{SEX}$$

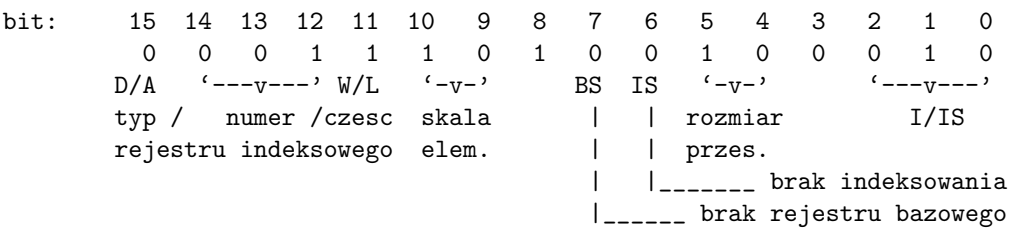
Pozostałe informacje, dotyczące drugiego słowa rozkazowego nie ulegają zmianie.
Przykładem zastosowania tego trybu może być tablica konwersji danych wpisana do kodu programu:

```
            ORG     ROM

* Deklaracja danych stalych                const char
TabPierw    DC.B    1,2,3,5,7,11              TabPierw[6] = {1,2,3,5,7,11}

* Inicjacja indeksu w tablicy
            MOVE.W  #5,D1                     NumerLiczby = 5;

* Odczyt liczby z tablicy
            MOVE.B  TabPierw(PC,D1.W),D0      Wynik = TabPierw[NumerLiczby];
```

W procesorach począwszy od MC68020 wprowadzono skalowanie indeksu w tym trybie na identycznej zasadzie, jak w trybie indeksowym z rejestrem adresowym.

1.5 Rozszerzenia trybów adresowania w rodzinie M68K

Już przy omawianiu trybów indeksowych okazało się, że procesor MC68020 wniósł istotne rozszerzenia nie tylko w dziedzinie sprętowej, ale również do adresowania. Przedstawimy teraz cztery indeksowe tryby adresowania, które pojawiły się w tym procesorze i w rzeczywistości dostarczyły kilku trybów pochodnych.
Wróćmy do drugiego słowa rozkazowego w trybach indeksowych. Bit b_8 (dotychczas zawsze ustawiony na wartość "0") stał się flagą wyróżniającą nowe tryby. W przypadku ustawienia tego bitu na "1", dolny bajt drugiego słowa rozkazowego przestaje być traktowany jako 8-bitowe przesunięcie i otrzymuje nową rolę:



Jak widać, możliwe jest wyłączenie rejestru bazowego, rejestru indeksowego, wybór wielkości przesunięcia bazowego (w tym wielkości zerowej, nie wymagającej dalszych słów rozkazowych) i wreszcie – wybór trybu indeksowania. Znaczenie poszczególnych opcji będzie jaśniejsze po omówieniu podstawowych odmian pośrednich trybów indeksowych.

1.5.1 Tryb adresowania pośredniego przez pamięć z postindeksacją (*memory indirect postindexed*)

Tym razem zaczniemy od algorytmu tworzenia adresu efektywnego:

$$EA = ((An) + bd_{SEX}) + (Xm)_{SEX} \star SCALE + od_{SEX}$$

Adres wynikowy składa się tu z:

- zawartości 32-bitowego słowa zaadresowanego przez zawartość rejestru bazowego z przesunięciem *base displacement* : $((An) + bd_{SEX})$
- zawartości rejestru indeksowego ze skalowaniem : $(Xm) \star SCALE$
- zewnętrznego przesunięcia *outer displacement* : od_{SEX}

Zauważmy, że ten tryb adresowania składa się z dwóch omawianych wcześniej trybów połączonych kaskadowo. Tryb pośredni rejestru adresowego z przesunięciem służy do pobrania z pamięci adresu bazowego dla trybu indeksowego. Zewnętrzne przesunięcie odnosi się do tego właśnie trybu. W ten złożony sposób można się odwołać w jednym adresie wynikowym do:
wybranego ($Xm, SCALE$) elementu tablicy będącej polem (bd_{SEX}) struktury wskazywanej przez wskaźnik zawarty w polu (od_{SEX}) struktury wskazywanej przez zawartość rejestru bazowego (An)
.

Rozbudujmy wcześniejszy przykład:

```
* Definicja struktury pomiar_t
            ORG     0
                                    typedef struct{
Datas       DS.B    Data_L              data_t Datas;
Temp        DS.L    24                  long Temp[24];
*                                   } pomiar_t;
Pomiar_L
*                                   Pomiar_L = sizeof(pomiar_t);

* Definicja struktury pacjent_t
            ORG     0
                                    typedef struct{
Nazwisko    DS.B    20                  char Nazwisko[20];
Imie        DS.B    20                  char Imie[20];
Badanie_p   DS.L    1                   pomiar_t *Badanie_p;
*                                   } pacjent_t;
Pacjent_L
*                                   Pacjent_L = sizeof(pacjent_t);

* Deklaracje zmiennych
            ORG     RAM
*                                   pacjent_t *ptr; /* w rejestrze A0 */
*                                   int NumerPom;  /* w rejestrze D1 */
*                                   long Wynik;    /* w rejestrze D0 */
Pacjent     DS.B    Pacjent_L       pomiar_t Pacjent;

* Dostep do elementu tablicy wewnatrz struktury
            ORG     ROM

* Inicjacja wskaznika struktury zewnetrznej
            MOVEA.L #Pacjent,A0     ptr = &Pacjent;

* Inicjacja indeksu w tablicy wewnetrznej
            MOVE.W  #12,D1          NumerPom = 12;

* odczyt pomiaru z tablicy
            MOVE.L  ([Badanie,A0],D1.W*4,Temp),D0
*                                   Wynik = ptr->Badanie->Temp[NumerPom];
```

Jak widać, w pojedynczym adresie docieramy do elementu tablicy podwójnie zagłębionej w strukturach. Nazwa "postindeksacja" wiąże się z kolejnością złożenia trybów składowych (najpierw pośredni, potem indeksowy).

1.5.2 Tryb adresowania pośredniego przez pamięć z preindeksacją (*memory indirect preindexed*)

Odwrócenie kolejności trybów składowych daje:

$$EA = ((An) + bd_{SEX} + (Xm)_{SEX} \star SCALE) + od_{SEX}$$

Tak możemy się odwołać do:
pola (od_{SEX}) struktury wskazywanej przez wskaźnik będący wybranym ($Xm, SCALE$) elementem tablicy będącej polem (bd_{SEX}) struktury wskazywanej przez zawartość rejestru bazowego (An) .
Oto ilustracja:

```
* Definicja struktury data_t
            ORG     0
                                    typedef struct{
Dzien       DS.B    1                   char Dzien;
Miesiac     DS.B    1                   char Miesiac;
Rok         DS.W    1                   short Rok;
*                                   } data_t;
Data_L
*                                   Data_L = sizeof(data_t);

* Definicja struktury wizyty_t
            ORG     0
                                    typedef struct{
Nazwisko    DS.B    20                  char Nazwisko[20];
Imie        DS.B    20                  char Imie[20];
DniWiz      DS.L    30                  data_t * DniWiz[30];
*                                   } wizyty_t;
Wizyty_L
*                                   Wizyty_L = sizeof(wizyty_t);

* Deklaracje zmiennych
            ORG     RAM
*                                   wizyty_t *ptr; /* w rejestrze A0 */
*                                   int NumerWiz;  /* w rejestrze D0 */
*                                   char DzienW;    /* w rejestrze D1 */
Wizyty      DS.B    Wizyty_L        wizyty_t Wizyty;

            ORG     ROM

* Inicjacja wskaznika struktury zewnetrznej
            MOVEA.L #Wizyty,A0      ptr = &Wizyty;

* Inicjacja indeksu w tablicy zewnetrznej
            MOVE.W  #12,D0          NumerWiz = 3;

* odczyt pomiaru z tablicy
            MOVE.L  ([DniWiz,A0,D0.W*4],Dzien),D0
*                                   DzienW = ptr->DniWiz[NumerWiz]->Dzien;
```

W opisanym trybie adresowanie indeksowe jest wykonywane w celu znalezienia w pamięci adresu pośredniego, stąd nazwa "preindeksacja".
W obu powyższych trybach występują dwa stałe przesunięcia: wewnętrzne (bazowe) i zewnętrzne. Są one umieszczane po drugim słowie rozkazu w takiej właśnie kolejności. Jak wspomniano wcześniej, jest wiele wariantów trybów adresowania za pośrednictwem pamięci, które wynikają z pominięcia niektórych składników

adresu wynikowego. W odniesieniu do podanych wcześniej elementów dolnego bajtu drugiego słowa rozkazowego warianty te zawarte są w tabelach:

Pole	Definicja
BS	wykluczenie rejestru bazowego *Base register Suppress* 0 = jest rejestr bazowy 1 = brak rejestru bazowego
IS	wykluczenie indeksu *Index Suppress* 0 = jest indeksowanie 1 = brak indeksowania
BD SIZE	rozmiar przesunięcia bazowego *Base Displacement SIZE* 00 = zarezerwowane 01 = zerowe przesunięcie 10 = 16-bitowe przesunięcie 11 = 32-bitowe przesunięcie
I/IS	wybór podtrybów *Index/Indirect Selection* wraz z polem IS opisane w następnej tablicy

IS	I/IS	Opis trybu
0	000	brak odwołania pośredniego
0	001	preindeksacja bez zewnętrznego przesunięcia
0	010	preindeksacja z 16-bitowym zewnętrznym przesunięciem
0	011	preindeksacja z 32-bitowym zewnętrznym przesunięciem
0	100	zarezerwowane
0	101	postindeksacja bez zewnętrznego przesunięcia
0	110	postindeksacja z 16-bitowym zewnętrznym przesunięciem
0	111	postindeksacja z 32-bitowym zewnętrznym przesunięciem
1	000	brak odwołania pośredniego
1	001	pośrednie bez zewnętrznego przesunięcia
1	010	pośrednie z 16-bitowym zewnętrznym przesunięciem
1	011	pośrednie z 32-bitowym zewnętrznym przesunięciem
1	100–111	zarezerwowane

1.5.3 Tryby adresowania względem licznika rozkazów za pośrednictwem pamięci

Analogicznie do trybu indeksowego z licznikiem rozkazów w miejsce bazowego rejestru adresowego wprowadzono dwa kolejne, złożone tryby adresowania za pośrednictwem pamięci:

- *program counter memory indirect postindexed*

$$EA = ((PC) + 2 + bd_{SEX}) + (Xm)_{SEX} \star SCALE + od_{SEX}$$

- *program counter memory indirect preindexed*

$$EA = ((PC) + 2 + bd_{SEX} + (Xm)_{SEX} \star SCALE) + od_{SEX}$$

Uzyskano je przez zastąpienie bazowego rejestru adresowego licznikiem rozkazów. Mają one podobne własności jak omówione tryby pośrednie przez pamięć z uwzględnieniem uwag wynikających z użycia zawartości licznika rozkazów jako adresu bazowego.

1.6 Przykładowy listing asemblacji

```
00000000  =00F00000                1  ROM         EQU     $f00000
00000000  =00001000                2  RAM         EQU     $1000
00000000                           3
00000000                           4  * Tryb po/sredni rejestru adresowego
00000000                           5
00000000  =000055AA                6  Dane        EQU     $55AA
00000000  =00FFF000                7  PortAdd     EQU     $FFF000
00000000                           8
00000000                           9  * Deklaracje zmiennych
00000000                          10  *
00000000                          11
00000000                          12  * Inicjacja wskaznika
00F00000                          13              ORG     ROM
00F00000  207C 00FFF000           14              MOVEA.L #PortAdd,A0
00F00006                          15
00F00006                          16  * Przeslanie danych do portu
00F00006  30BC 55AA               17              MOVE.W  #Dane,(A0)
00F0000A                          18
00F0000A                          19
00F0000A                          20  * Tryb po/sredni rejestru adresowego
00F0000A                          21  * (address register indirect with po
00F0000A                          22
00F0000A                          23  * Deklaracje zmiennych
00001000                          24              ORG     RAM
00001000                          25  *
00001000                          26  *
00001000                          27  TablA       DS.W    100
000010C8                          28  TablB       DS.W    100
00001190                          29
00001190                          30  * Inicjacja wskaznikow
00F00000                          31              ORG     ROM
00F00000  207C 00001000           32              MOVEA.L #TablA,A0
00F00006  227C 000010C8           33              MOVEA.L #TablB,A1
00F0000C                          34
00F0000C                          35  * Przepisanie elementow i przesuniec
00F0000C  32D8                    36              MOVE.W  (A0)+,(A1)+
00F0000E                          37
00F0000E                          38
00F0000E                          39  * Tryb po/sredni rejestru adresowego
00F0000E                          40  * (address register indirect with pr
00F0000E                          41
00F0000E                          42  * Przesuniecie wskaznikow i przepisa
00F0000E  3320                    43              MOVE.W  -(A0),-(A1)
00F00010                          44
00F00010                          45
00F00010                          46  * Tryb po/sredni rejestru adresowego
00F00010                          47  * (address register indirect with di
00F00010                          48
00F00010                          49  * Definicja struktury data_t
00000000                          50              ORG     0
00000000                          51  *
00000000                          52  Dzien       DS.B    1
```

```
00000001                          53  Miesiac     DS.B    1
00000002                          54  Rok         DS.W    1
00000004                          55  *
00000004                          56  Data_L
00000004                          57  *
00000004                          58
00000004                          59  * Deklaracje zmiennych
00001000                          60              ORG     RAM
00001000                          61  *
00001000                          62  Data        DS.B    Data_L
00001004                          63
00001004                          64  * Dostep do wnetrza struktury
00F00000                          65              ORG     ROM
00F00000  207C 00001000           66              MOVEA.L #Data,A0
00F00006  317C 07CC 0002          67              MOVE.W  #1996,Rok(A0)
00F0000C                          68
00F0000C                          69
00F0000C                          70  * Tryb po/sredni rejestru adresowego
00F0000C                          71  * (address register indirect with in
00F0000C                          72
00F0000C                          73  * Definicja struktury pomiar_t
00000000                          74              ORG     0
00000000                          75  *
00000000                          76  Datas       DS.B    Data_L
00000004                          77  Temp        DS.B    24
0000001C                          78  *
0000001C                          79  Pomiar_L
0000001C                          80  *
0000001C                          81
0000001C                          82  * Deklaracje zmiennych
00001000                          83              ORG     RAM
00001000                          84  *
00001000                          85  *
00001000                          86  *
00001000                          87  Pomiar      DS.B    Pomiar_L
0000101C                          88
0000101C                          89  * Dostep do elementu tablicy wewnatr
00F00000                          90              ORG     ROM
00F00000                          91
00F00000                          92  * Inicjacja wskaznika struktury
00F00000  207C 00001000           93              MOVEA.L #Pomiar,A0
00F00006                          94
00F00006                          95  * Inicjacja indeksu w tablicy
00F00006  7200                    96              MOVE.L  #0,D1
00F00008                          97
00F00008                          98  * odczyt pomiaru z tablicy
00F00008  1030 1804               99              MOVE.B  Temp(A0,D1.L),D0
00F0000C                         100
```

2 Lista instrukcji M68000

Tworzenie programów w języku asemblera wymaga niewątpliwie znajomości pewnego podzbioru listy instrukcji procesora. Zapoznanie się z listą instrukcji MC68000 poprzedzimy omówieniem rejestru statusowego, a właściwie jego części flagowej. Niezbędne będzie również pewne wstępne objaśnienie konwencji zapisu przyjętych w typowych asemblerach dla procesorów Motoroli. W dalszym ciągu przedstawimy sposób działania poszczególnych instrukcji i dopuszczalne dla nich tryby adresowania (poznane wcześniej). Informacje te powinny umożliwić tworzenie pierwszych programów.

2.1 Rejestr statusowy (*CCR*)

Rozkazy wykonywane przez procesor, poza właściwymi sobie operacjami na rejestrach i pamięci, modyfikują poszczególne bity rejestru statusowego (*SR – Status Register*) procesora. Przy omawianiu listy rozkazów MC68000 skupimy się na jego części dolnej, zwanej rejestrem flagowym (*CCR – Condition Code Register*), która jest dostępna niezależnie od trybu pracy procesora (*User/Supervisor*).
CCR zawiera pięć flag:

```
bit:       7   6   5   4   3   2   1   0
flaga:     -   -   -   X   N   Z   V   C
                       |   |   |   |   |__ Carry    - przeniesienie
                       |   |   |   |______ Overflow - nadmiar
                       |   |   |__________ Zero     - wynik zerowy
                       |   |______________ Negative - wynik ujemny
                       |__________________ Extend   - przedluzenie
```

Flaga `Z` jest ustawiana na "1" gdy wynik ostatnio wykonanej operacji jest zerowy, to znaczy wszystkie aktywne bity wyniku są zerowe (ilość bitów aktywnych zależy od typu danych użytych w operacji). W przeciwnym przypadku flaga ta jest zerowana.
Flaga `N` jest ustawiana na "1" gdy wynik operacji jest ujemny, w przeciwnym przypadku jest zerowana. Ponieważ w procesorze MC68000 używana jest reprezentacja liczb ze znakiem typu uzupełnienia do 2, więc flaga `N` jest kopią najstarszego z aktywnych bitów wyniku.
Flaga `C` jest ustawiana na "1", gdy w wyniku operacji arytmetycznej następuje przeniesienie z najstarszej aktywnej pozycji, lub przy operacji przesywania – wysuwany jest bit "1".
Flaga `X` jest ustawiana podobnie jak `C`, ale tylko przez niektóre instrukcje. Pełni rolę bitu przeniesienia potrzebnego do przedłużania operacji arytmetycznych na większą ilość słów.
Flaga `V` sygnalizuje wystąpienie nadmiaru w czasie ostatniej operacji arytmetycznej. Nadmiar ma miejsce, gdy wynik operacj nie mieści się w przyjętej reprezentacji liczb. Na przykład, po dodaniu dwóch liczb ujemnych otrzymujemy wynik dodatni.
Opisana interpretacja bitów rejestru flagowego odnosi się do typowych sytuacji, testowanych po operacjach arytmetycznych i logicznych. W ogólności interpretacja tych bitów może być specyficzne dla niektórych instrukcji.
Wpływ danej instrukcji na daną flagę może być określony na jeden z pięciu sposobów (w dalszym ciągu będziemy powyższych oznaczeń używać przy opisywaniu rozkazów):

- brak wpływu (co będziemy oznaczać jako `-`)
- zmiana warunkowa (oznaczana dalej jako `!`)
- wpływ nieokreślony (`?`)
- zmiana bezwarunkowa na "0" (`0`)
- zmiana bezwarunkowa na "1" (`1`)

2.2 Notacja asemblerowa

Zapis programu w postaci tekstu źródłowego wymaga przestrzegania pewnych podstawowych reguł składni i znajomości niektórych dyrektyw (pseudoinstrukcji) rozumianych przez program tłumaczący (asembler) nasze źródło na język maszynowy (kod binarny).
Program źródłowy w języku asemblera składa się z linii. W odróżnieniu od wielu języków wyższego rzędu, język asemblera ma składnię wrażliwą na tzw. *białe znaki* (przerwy, tabulacje). Wynika to z faktu, że te właśnie znaki służą do oddzielania poszczególnych pól, z których składa się każda linia programu.
Typowa linia asemblera Motoroli składa się z następujących pól:

```
etykieta    rozkaz  argument(y)          komentarz

* Przyklad linii zrodlowych asemblera

INIT        MOVE.B  #$FC05,PORT1         zainicjuj wyjscia
            RTS                          koniec podprogramu
```

Oto podstawowe reguły dotyczące redakcji linii:

- Argumenty nie mogą być oddzielone przerwami, lecz muszą stanowić jedno pole. W instrukcjach wieloargumentowych oddziela się je przecinkami.
- Dowolny ciąg białych znaków jest traktowany identycznie jak pojedynczy biały znak.
- Nazwa rozkazu może być uzupełniona, jeśli to ma sens, rozmiarem danych. Używa się: `.B` dla bajtu, `.W` dla słowa i `.L` dla długiego słowa. Domyślny jest rozmiar słowa.
- Pole etykiety uważa się za puste, jeżeli pierwszy znak w linii jest biały.
- Dopuszczalne są linie zawierające tylko pole etykiety.
- Pierwszym znakiem nazwy symbolicznej (etykiety) musi być litera.
- Pewne etykiety są zastrzeżone jako nazwy rejestrów prcesora:
  - D0–D7 – rejestry danych
  - A0–A7 – rejestry adresowe
  - SP, USP – wskaźniki stosu
  - SR, CCR – rejestr statusowy (flagowy)
  - PC – licznik rozkazów
- Znak "`*`" w pierwszej kolumnie linii kodu źródłowego oznacza, że cała linia stanowi komentarz (ignorowany przez asembler). Puste linie są również ignorowane.
- Pole komentarza kończy się wraz z końcem linii.
- W liniach zawierających rozkazy bezargumentowe pole komentarza występuje bezpośrednio po polu rozkazu.

Stałe liczbowe używane w tekście źródłowym zapisuje się w notacji przedrostkowej. Znak określający podstawę systemu liczbowego jest umieszczony bezpośrednio przed cyframi liczby:

podstawa	przedrostek	zapis	dopuszczalne cyfry
10		`197`	0,1,2,3,4,5,6,7,8,9
2	`%`	`%11000101`	0,1
8	`@`	`@305`	0,1,2,3,4,5,6,7
16	`$`	`$C5`	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Stałe tekstowe mogą być umieszczane jako ciągi znaków ujęte w apostrofy:

```
MOVE.W      #'UL',D0          wpisz $554C do rejestru D0
```

Większość asemblerów dopuszcza stosowanie prostych wyrażeń arytmetycznych do zadawania stałych:

```
MOVE.L      #KWARC/16,D0      wpisz czestotliwosc do D0
```

Dopuszczalne bywają operacje dodawania (`+`), odejmowania (`-`), mnożenia (`*`) i dzielenia całkowitego (`/`), logiczne operatory bitowe (`&` - and, `|` - or) i grupowanie w nawiasach. Ze względu na różnorodność implementacji asemblerów należy zachować daleko idącą ostrożność i (w przypadku korzystania z nowych narzędzi) sprawdzić ich możliwości przed poważnym użyciem.
Podstawowymi dyrektywami asemblera są:

- `ORG` – ustawienie licznika adresowego na wartość zadaną w polu argumentu
- `EQU` – przypisanie nazwie w polu etykiety wartości z pola argumentu
- `DS.s` – przestawienie licznika adresowego (zarezerwowanie miejsca na określoną w polu argumentu ilość słów o długości wynikającej z rozmiaru danych `.s`)
- `DC.s` – wytworzenie stałej o rozmiarze `.s` i wartości podanej w argumencie

Powyższy opis powinien wystarczyć do zrozumienia przykładów ilustrujących listę rozkazów. Pewne przykłady wystąpiły już w trakcie omawiania trybów adresowania. Inne dyrektywy asemblera i rozszerzenia składni będą wprowadzane w w trakcie wykładu w miarę potrzeby.

2.3 Kody instrukcji MC68000

Procesor MC68000 ma 56 instrukcji, które wraz z opisanymi wcześniej trybami adresowania dają programiście bardzo duże możliwości. W celu usystematyzowania ich prezentacji podzielono je na sześć grup:

- instrukcje przesyłania danych
- operacje adresowe
- instrukcje arytmetyczne
- instrukcje logiczne
- instrukcje sterujące przebiegiem programu
- instrukcje uprzywilejowane

Tryby adresowania również wygodnie jest podzielić na grupy pod kątem ich dopuszczalności w różnych rozkazach:

- D/A – bezpośrednie tryby rejestrowe
- –d(A,X)+ – pośrednie tryby rejestrowe i indeksowe
- abs – bezwzględne tryby adresowe
- d(PC,X) – względem licznika rozkazów
- imm – dane natychmiastowe

2.3.1 Instrukcje przesyłania danych

Instrukcje przesyłania danych służą do przemieszczania danych pomiędzy komórkami pamięci i rejestrami procesora. Należą do nich takie rozkazy, jak: `MOVE`, `CLEAR`, `EXG`, `SWAP`, `MOVEM`, `MOVEQ`, `MOVEP`.
Rozkazy typu `MOVE` przepisują zawartość jednej komórki pamięci do drugiej, komórki pamięci do rejestru, rejestru do komórki pamięci, lub rejestru do innego rejestru. Rozmiar danych może obejmować bajty (`MOVE.B`), słowa 16-bitowe (`MOVE.W` lub `MOVE`), długie (32-bitowe) słowa (`MOVE.L`). Ogólna postać tego rozkazu to:

```
     B
MOVE.W      <skad>,<dokad>
     L
```

Wykonywana operacja:

$$(< skad >) \rightarrow < dokad >$$

Specyfikacja rozmiaru danych dotyczy zarówno źródła, jak i przeznaczenia. Operacja `MOVE` warunkowo ustawia flagi `N` i `Z` w zależności od przesyłanych danych (jeśli wszystkie bity danych są równe "0", to flaga Z = 1, jeśli najstarszy bit – bit znaku jest równy "1", to flaga N = 1). Flaga `X` nie jest zmieniana, flagi `V` i `C` są bezwarunkowo zerowane.
Tryby adresowania, w których można wyrazić źródło i miejsce przeznaczenia przesyłanych danych dopełniają specyfikację instrukcji `MOVE`:

```
              D/A          D/A
     B      -d(A,X)+     -d(A,X)+        X N Z V C
MOVE.W        abs     ,    abs           - ! ! 0 0
     L      d(PC,X)        ---
              imm          ---
```

Zwróćmy uwagę, że w spesyfikacji źródła mogą się pojawić dowolne tryby adresowe, a w specyfikacji adresu przeznaczenia brakuje dwóch klas. Mimo to dla procesora MC6800 mamy 288 różnych form tej instrukcji.
Ponieważ liczne przykłady zastosowań instrukcji `MOVE` podano przy prezentacji trybów adresowania, teraz je pominiemy.
Następny rozkaz, `CLR`, zeruje bajt, słowo, lub długie słowo. Nie wymaga on adresu źródłowego, gdyż z góry przesądzoną wartścią danych jest "0":

```
                    D/A
    B             -d(A,X)+        X N Z V C
CLR.W               abs           - 0 1 0 0
    L               ---
                    ---
```

Rozkaz `EXG` działa tylko pomiędzy rejestrami procesora. Pozwala on szybko wymieniać wzajemnie ich zawartość:

$$(Rn) \leftrightarrow Rm$$

Klasycznie potrzebna jest dodatkowa lokalizacja robocza (tymczasowa), by przy pomocy przepisywania uzyskać efekt opisany powyżej:

$$(Rn) \rightarrow T$$
$$(Rm) \rightarrow Rn$$
$$(T) \rightarrow Rm$$

Operacja ta nie narusza żadnej flagi w CCR:

```
              D/A          D/A
              ---          ---          X N Z V C
EXG.L         ---     ,    ---          - - - - -
              ---          ---
              ---          ---
```

Rozkaz powyższy może być użyty do ustawienia nowej wartości rejestru z równoczesnym zachowaniem dotychczasowej (przydatne przy szybkim przełączaniu kontekstu w małych systemach wielozadaniowych).
Rozkaz `SWAP` operuje na jednym rejestrze. Jego działanie polega na zamianie miejscami górnego (bity $b_{31} - b_{16}$) i dolnego ((bity $b_{15} - b_0$) słowa tego rejestru. Jest szczególnie przydatny, jeśli operujemy na danych 16-bitowych, bo pozwala podwoić ilość miejsca dostępnego dla programisty w rejestrach porcesora. Ustawia flagi `Z` i `C` w zależności od wyniku (całego rejestru):

```
                          Dn
                          ---          X N Z V C
SWAP                      ---          - ! ! 0 0
                          ---
                          ---
```

Przykład zastosowania:

```
MOVE.W      PORT,D0     zaladuj starsze slowo
SWAP        D0          przestaw je na gorne bity
MOVE.W      PORT,D0     zaladuj mlodsze slowo
```

Instrukcja `MOVEM` (*MOVE Multiple*) zasygnalizowana wcześniej, służy do przesyłania zawartości wybranego podzbiory rejestrów adresowych i danych procesora z lub do pamięci. W praktyce programowania wielokrotnie zachodzi potrzeba tymczasowego przechowania zawartości rejestrów roboczych (przy podprogramach, procedurach obsługi przerwań, przełączaniu kontekstu w systemach wielozadaniowych itp.). Zamiast wykonywać kilka instrukcji `MOVE.L` można użyć `MOVEM` z argumentem będącym zakodowaną listą rejestrów biorących udział w przesłaniu. W zależności od tego, czy lista rejestrów stanowi specyfikację "`<skad>`", czy "`<dokad>`", mamy do czynienia z zachowywaniem, lub odtwarzaniem grupy rejestrów:

```
                          -d(A,X)+      X N Z V C
MOVEM.W    <lista_rej> ,   abs          - - - - -
     .L

MOVEM.L     <lista_rejestrow>,<dokad>
```

Listę rejestrów zadaje się w tekście źródłowym przez podanie zakresów (`Ri-Rj`) i/lub wyliczenie rejestrów (`RiŔjŔk`). Kolejność specyfikacji rejestrów nie odgrywa roli, gdyż asembler koduje tak zadany zestaw rejestrów w jednym słowie 16-bitowym, w którym każdemu rejestrowi procesora odpowiada konkretny bit:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0

rejestr: A7  A6  A5  A4  A3  A2  A1  A0  D7  D6  D5  D4  D3  D2  D1  D0
```

Rejestry są przesyłane poczynając od najniższych bitów słowa kodowego do kolejnych komórek pamięci, poczynając od wynikowego adresu przeznaczenia. W przypadku przeciwnego kierunku (odtwarzanie zawartości rejestrów):

```
                 -d(A,X)+                  X N Z V C
MOVEM.W            abs    ,  <lista_rej>   - - - - -
     .L          d(PC,X)
```

rejestry są odtwarzane z kolejnych komórek pamięci w tej samej kolejności.
Wyjątek stanowi prrzypadek, gdy do zaadresowania miejsca przeznaczenia zastosowano tryb z autopredekrementacją. Wtedy mamy do czynienia z sytuacją, gdy adres pochodzący z rejestru adresowego wskazuje na wierzchołek stosu. Kolejność rejestrów w kodzie maski jest w tym przypadku odwrócona:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0

rejestr: D0  D1  D2  D3  D4  D5  D6  D7  A0  A1  A2  A3  A4  A5  A6  A7
```

by zachować podaną wcześniej kolejność rozmieszczenia rejestrów w pamięci (w trybie z autodekrementacją kolejno generowane adresy maleją, a po przesłaniu zawartość rejestru adresowego wskazuje na miejsce umieszczenia ostatnio składowanego rejestru). Pozwala to zachować zgodność (a właściwie przeciwstawność) operacji:

```
* zachowanie rejestrow na stosie
          MOVEM.L     A0/A1/A5-A6/D0/D6,-(A7)
* tu :  operacje korzystajace z zachowanych rejestrow

* odtworzenie rejestrow ze stosu
          MOVEM.L     (A7)+,A0/A1/A5-A6/D0/D6
```

Taka para operacji składa na stosie rejestry `A0,A1,A5,A6,D0,D6` procesora by po wykonaniu operacji (prawdopodobnie niszczących ich zawartość jako rejestrów roboczych) odtworzyć ich pierwotny stan.
Rozmiar danych użytych w operacji może wynosić 16 lub 32 bity (`.W` lub `.L`). O ile nie ma wątpliwości przy odtwarzaniu rejestrów w drugim przypadku, to rozmiar słowa wymaga komentarza. Przy składowaniu do pamięci wpisywana jest zawartość dolnych 16 bitów rejestru. Przy odtwarzniu – cały rejestr (32 bity) jest wypełniany znakowym rozszerzeniem 16-bitowego słowa z pamięci. Powoduje to, że cały 32-bitowy rejastr uzyskuje zawartość równą (jako liczba ze znakiem) wcześniej składowanej 16-bitowej części.
Instrukcja `MOVEQ` (*MOVE Quick*) pozwala szybko ustawić rejestr danych na wartość dającą się przedstawić (jako liczba ze znakiem) na 8-bitowym bajcie. Dane te mieszczą się w słowie rozkazowym, co daje zarazem zwięzłość kodu i szybkość jego działania.

```
                                          X N Z V C
MOVEQ          #dane ,  Dn                - ! ! 0 0
```

W tej instrukcji domyślnym (i zarazem jedynym)rozmiarem danych jest `.L`, ponieważ cały rejestr (32 bity) jest modyfikowany.
Instrukcja `MOVEP` (*MOVE Peripheral*) powstała w MC68000 dla ułatwienia komunikacji z 8-bitowymi portami przez 16-bitową magistralę procesora.

```
                  Dn   , d(An)            X N Z V C
MOVEP.W          d(An) ,  Dn              - - - - -
      .L
```

W zależności od rozmiaru danych (`.W`, `.L`) przesyła ona kolejne dwa (cztery) bajty składowe słowa (długiego słowa) zawartego w wyspecyfikowanym rejestrze danych do **kolejnych, co drugich** bajtów w pamięci, począwszy od adresu `d(An)`. Jest to korzystne w przypadku, gdy 8-bitowe urządzenie zewnętrzne jest przyłączone do 16-bitowej magistrali danych tylko na jednej z jej połówek.
Przeciwny kierunek przesyłania pozwala złożyć w rejestrze danych słowo lub długie słowo z odczytanych z jednej połówki 16-bitowej magistrali danych bajtów.
Odczyt wartości flag z rejestru CCR (*Condition Code Register*) i modyfikacja tego rejestru jest możliwa dzięki specjalnym odmianom instrukcji `MOVE`:

```
                        Dn
                     -d(An)+              X N Z V C
MOVE.W          CCR ,  abs                - - - - -
                 SR    ---
                       ---


                  Dn
                -d(An)+                   X N Z V C
MOVE.W            abs  , CCR              ! ! ! ! !
                d(PC,X)   SR
                  imm
```

Analogiczne operacje dla całego rejestru statusowego `SR` (*Status Register*) są dozwolone tylko w trybie uprzywilejowanym (*supervisor*). Tylko w procesorze MC68000 można odczytać SR w trybie normalnym (*usr*), w następnych procesorach z rodziny M68K zarówno odczyt, jak i zapis traktowane są jako naruszenie uprzywilejowania (*privillege violation*) i powoduje odpowiednią akcję awaryjną (*exception*).

2.3.2 Operacje adresowe

Instrukcje służące do operowania na adresach nie modyfikują zawartości rejestru statusowego. Do grupy tej należą: `MOVEA`, `ADDA`, `SUBA`, `LEA`, `PEA`. Najczęściej (poza `PEA`) miejscem przeznaczenia danych jest któryś z ośmiu rejestrów adresowych procesora.
Rozkaz `MOVEA` jest w procesorze M68K kodowany tak, jak byłby kodowany rozkaz `MOVE` z rejestrem adresowym jako miejscem docelowym, różni się jednak dość istotnie działaniem.

```
                D/A          An
              -d(A,X)+       ---          X N Z V C
MOVEA.W         abs     ,    ---          - - - - -
      L      d(PC,X)         ---
                imm          ---
```

Z oczywistych względów rozmiar danych nie może być bajtem (). W przypadku rozmiaru `.W` zmodyfikowany zostaje cały rejestr docelowy tak, by jego zawartość była równa danej wejściowej w sensie liczby ze znakiem (U2). Oznacza to, że dane wejściowe podlegają opisanej wcześniej operacji rozciągnięcia bitu znakowego na starsze słowo rejestru docelowego. Jak wspomniano wcześniej, żadne flagi nie są modyfikowane.
Dla wygody programisty asemblery automatycznie tłumaczą rozkazy zapisane:

```
MOVE    <ea>,An
```

jako:

```
MOVEA   <ea>,An
```

Rozkaz `ADDA` służy do dodawania liczby zadanej przez podanie specyfikacji adresu wynikowego do rejestru adresowego. Podobnie jak w przypadku instrukcji `MOVEA`, flagi nie są modyfikowane:

```
                D/A          An
              -d(A,X)+       ---          X N Z V C
ADDA.W          abs     ,    ---          - - - - -
     L        d(PC,X)         ---
                imm          ---
```

W przypadku danych o rozmiarze `.W` dokonywane jest rozciągnięcie znaku, a w operacji bierze udział cały rejestr adresowy.
Analogicznie (z dokładnością do znaku operacji) działa rozkaz odejmowania: `SUBA`.
Instrukcja `LEA` (*Load Effective Address*) wspomniana już wcześniej, służy do załadowania do rejestru adresowego (jak poprzednio, miejscem docelowym jest cały rejestr adresowy) adresu wynikowego wyliczonego w trakcie wykonywania programu na podstawie trybu adresowania argumentu źródłowego:

```
                ---          An
              d(A,X)         ---          X N Z V C
LEA.L           abs     ,    ---          - - - - -
              d(PC,X)         ---
                ---          ---
```

Adres źródłowy musi się odnosić do pamięci, niedopuszczalne są tryby z automodyfikacjami.
Rozważmy różnice pomiędzy tym rozkazem, a `MOVEA` w trybie natychmiastowym.
Poniższe przykłady, różniące się tylko sposobem inicjacji rejestru bazowego, mają istotnie różne własności.
W pierwszej wersji kod wynikowy (binarny) programu będzie pracował poprawnie tylko wtedy, gdy tablica znaków opatrzona etykietą "`Napis`" będzie w miejscu, dla którego program został zasemblowany. Wynika to z faktu, że wartość adresu jest wpisana jako stała do wnętrza kodu programu (argument natychmiastowy instrukcji `MOVEA`). W rezultacie, ponieważ tablica ta znajduje się w sekcji kodu, program nie będzie mógł zostać przeniesiony w inne miejsce bez modyfikacji kodu.
Przy zastosowaniu trybu natychmiastowego dla instrukcji `MOVEA` w celu wpisania adresu początku tablicy do rejestru A0 mamy:

```
* chcemy wys/la/c napis kolejno, znak po znaku, do portu

Napis       DC.B    'Ala ma kota'
            DC.B    0
*
* Inicjacja rejestru bazowego, wersja 1
            MOVEA.L #Napis,A0
*
* Dost/ep do kolejnych znakow napisu
            MOVE.B  (A0)+,Port
```

Ananlogiczny fragment kodu wykorzystujący do zainicjowania rejestru adresowego przy pomocy trybu adresowania względem licznika rozkazów i instrukcji `LEA` wygląda następująco:

```
* chcemy wys/la/c napis kolejno, znak po znaku, do portu

Napis       DC.B    'Ala ma kota'
            DC.B    0
*
* Inicjacja rejestru bazowego, wersja 2
            LEA.L   Napis(PC),A0
*
* Dost/ep do kolejnych znakow napisu
            MOVE.B  (A0)+,Port
```

Druga wersja jest całkowicie niezależna od położenia kodu binarnego w pamięci. Tu adres wynikowy "`Napis`"jest wyliczany względem licznika rozkazów w chwili wykonywania instrukcji `LEA`, a w kodzie programu znajduje się tylko adres względny (przesunięcie adresu tablicy względem adresu, pod którym umieszczona jest instrukcja `LEA`). Dzięki temu, przy przemieszczeniu programu w pamięci, adres będzie odzwierciedlał aktualne położenie napisu. Bez modyfikacji kodu można przemieszczać nasz program w dowolne miejsce bez zmiany jego działania. Zalety tej własności są oczywiste, jeśli rozważymy uruchamianie programów w środowisku wielozadaniowego systemu operacyjnego (np. OS–9), lub choćby prostego systemu jednozadaniowego, jakim jest DOS. Załadowany z dysku kod przesuwalny można uruchomić natychmiast po zakończeniu wczytywania do obszaru pamięci przydzielonego przez system. W przypadku tzw. kodu relokowalnego (np. programy typu `.EXE` w systemie DOS) procedura ładująca program z dysku (*relocating loader*) musi zmodyfikować wskazane w nagłówku adresy bezwzględne w zależności od fizycznego adresu, pod którym umieszczono kod.
Instrukcja `PEA` (*Push Effective Address*) jest bardzo podobna do `LEA`. Jedyna różnica polega na miejscu docelowym, w którym umieszczany jest wyliczony adres wynikowy. Zamiast (wymienionego w drugim argumencie `LEA`) rejestru adresowego użyte jest miejsce w pamięci wskazywane przez **aktualny** wskaźnik stosu z jego predekrementacją. Aktualny wskaźnik stosu jest zależny od trybu pracu procesora pokazywanego przez bit S w rejestrze statusowum. W trybie nadzorcy (*supervisor* - S=1) jest to wskaźnik stosu systemowego (*SSP*), a w trybie użytkownika (*user* - S=0) – stosu użytkownika (*USP*). W obu przypadkach tryb adresowania miejsca docelowego można określić przy pomocy rejestru A7 jako `-(A7)` lub przy pomocy nazwy mnemotechnicznej przyjętej dla wskaźnika stosu aktualnego `-(SP)`. Należy jednak pamiętać, że w zapisie asemblerowym **nie podaje się drugiego argumentu**, jest on domyślny:

```
                  ---
                 d(A,X)             X N Z V C
PEA.L              abs              - - - - -
                 d(PC,X)
                  ---
```

Typowym zastosowaniem instrukcji `PEA` jest składowanie na stosie argumentów bezpośrednio przed wywołaniem procedury przy przekazywaniu parametrów w postaci wskaźników (adresów). Wrócimy do tego tematu przy omawianiu procedur i łączenia procedur asemblerowych z językami wyższego rzędu.

2.3.3 Instrukcje arytmetyczne

Instrukcje arytmetyczne procesora M68K to:

- dodawanie (`ADD`, `ADDI`, `ADDQ`, `ADDX`)
- odejmowanie (`SUB`, `SUBI`, `SUBQ`, `SUBX`)
- mnożenie (`MULS`, `MULU`)
- dzielenie (`DIVS`, `DIVU`)
- operacje na liczbach dziesiętnych kodowanych binarnie (`ABCD`, `SBCD`, `NBCD`)
- przesunięcia arytmetyczne (`ASL`, `ASR`)
- operacje pomocnicze (`EXT`, `NEG`, `NEGX`).

Instrukcja dodawania `ADD` wymaga dwóch argumentów, a jej wynik musi być wysłany do pewnego miejsca docelowego. Jak wspomniano wcześniej, operacja ta jest 1.5-adresowa, co oznacza, że tylko jeden z jej argumentów może być zadany w (prawie) dowolnym trybie adresowania. Drugi z argumentów musi się znajdować w rejestrze danych:

```
ADD     <ea>,Dn             (<ea>) + (Dn) -> Dn
ADD     Dm,<ea>             (Dm) + (<ea>) -> <ea>
```

W pierwszym przypadku dopuszczalne są wszystkie tryby adresowania, ponieważ określają one dane, które są tylko czytane:

```
                D/A          Dn
    B        -d(A,X)+        ---           X N Z V C
ADD.W          abs     ,     ---           ! ! ! ! !
    L        d(PC,X)         ---
               imm           ---
```

W drugim przypadku można używać tylko trybów, które pozwalają na zapis wyniku (por. `MOVE`):

```
                Dn           Dm
    B           ---       -d(A,X}+         X N Z V C
ADD.W           ---     ,   abs            ! ! ! ! !
    L           ---         ---
                ---         ---
```

Wykonanie tej instrukcji powoduje ustawienie wszystkich bitów flagowych zgodnie z przebiegiem operacji i stanem jej wyniku. Rozmiar danych może wynosić 8 bitów (`.B`), 16 bitów (`.W` – wielkość domyślna) lub 32 bity (`>L`).
Instrukcja `ADDI` (*ADD Immediate*) pozwala dodawać stałą wartość (zadaną w trybie danych natychmiastowych jako pierwszy argument) do zawartości miejsca wskazanego przez wynikowy adres podany w drugim argumencie:

```
                ---          Dm
    B           ---       -d(A,X}+         X N Z V C
ADDI.W          ---     ,   abs            ! ! ! ! !
    L           ---         ---
                imm         ---
```

Jak widać, pozwala to na wykonywanie tej operacji na komórkach pamięci, np. w celu zwiększenia zawartości liczników:

```
            ADDI.W  #44,LICZ              44 + (LICZ) -> LICZ
*                                         LICZ += 44;
```

Bardzo często potrzebne jest zwiększenie zawartości licznika o małą wartość (w innych procesorach przewidziano do tego celu instrukcje inkrementacji, czyli zwiększania o 1). Jeśli licznik jest umieszczony w rejestrze, to procesor M68K pozwala go zwiększyć o wartość od 1 do 8. Służy do tego instrukcja `ADDQ` (*ADD Quick*):

```
            ADDQ  #4,D4                   4 + (D4) -> D4
```

Dzięki ograniczeniu wielkości argumentu natychmiastowego do trzech bitów uzyskano bardzo szybki i zwarty (jedno słwo kodu) rozkaz.

```
                    ---          Dm
         B          ---          ---           X N Z V C
      ADDQ.W        ---    ,     ---           ! ! ! ! !
         L          ---          ---
                    imm          ---
```

Istnieje on również w wersji dla rejestru adresowego:

```
                    ---          Am
         B          ---          ---           X N Z V C
      ADDQ.W        ---    ,     ---           - - - - -
         L          ---          ---
                    imm          ---
```

Różnica polega wyłącznie na sposobie działania na flagi (zgodnie z przyjętą konwencją operacje adresowe nie modyfikują flag).

```
            ADDQ  #4,A5                   4 + (A5) -> A5
```

Wprawdzie arytmetyka całkowitoliczbowa na liczbach o długości 32 bitów wydaje się wystarczająca, ale w szczególnych przypadkach trzeba używać większych liczb dla zachowania dokładności. Rozszerzenie dodawania na dłuższe liczby umożliwia instrukcja `ADDX` (*ADD eXtended*):

```
* pierwsza liczba 96-bitowa:  D1:D2:D3
* druga liczba 96-bitowa:     D4:D5:D6
* wynik 96-bitowy:            D4:D5:D6
            ADD.L   D3,D6                 (D3) + (D6) -> D6
            ADDX.L  D2,D5                 (D2) + (D5) + (CCR:X) -> D5
            ADDX.L  D1,D4                 (D1) + (D4) + (CCR:X) -> D4
```

Jak widać, dodawanie najmniej znaczących słów jest wykonywane **bez** uwzględniania bitu rozszerzenia (flaga X z rejestru CCR), a dalsze – z jego uwzględnieniem.
Dopuszczalne są tylko dwie formy adresowania:

```
         B                                    X N Z V C
      ADDX.W        Dn     ,    Dm            ! ! ! ! !
         L
```

oraz:

```
         B                                    X N Z V C
      ADDX.W       -(An)   ,   -(Am)          ! ! ! ! !
         L
```

Pierwsza jest oczywista, jak w podanym wyżej przykładzie. Wersja z autopredekrementacją służy do dodawania liczb o dowolnej precyzji umieszczonych na stosach roboczych, utworzonych przez programistę:

```
* wskaznik stosu, na kt/orym umieszczono pierwsza liczbe: A1
* wskaznik stosu, na kt/orym umieszczono druga liczbe:    A2
* wskaznik stosu, na kt/orym umieszczono bedzie wynik:    A3
            MOVEA.L A2,A3               zachowanie wska/znika stosu wyniku
            ADD.L   -(A1),-(A2)         dodanie najmniej znaczacych slow
* nastepna instrukcja w petli o dlugosci zaleznej od dlugosci liczb
            ADDX.L  -(A1),-(A2)         dodanie kolejnych slow
```

Zachowanie wskaźnika stosu wynikowego jest konieczne, bo tryb z autopredekrementacją modyfikuje A2 (i A1). Zwróćmy też uwagę na zgodność kolejności wykonywania operacji z leksykalną konwencją rozmieszczenia liczb w pamięci (a więc i na stosach).
Instrukcje odejmowania (`SUB`, `SUBI`, `SUBQ`, `SUBX`) służą do odejmowania pierwszego argumentu od zawartości miejsca wskazanego przez wynikowy adres podany w drugim argumencie. Poza znakiem wykonywanej na danych operacji nie różnią się niczym od omówionych wyżej instrukcji odejmowania (`ADD`, `ADDI`, `ADDQ`, `ADDX`).
Instrukcje mnożenia mają dwa warianty:

- `MULS` (*MULtiply Signed*) - mnożenie liczb ze znakiem
- `MULU` (*MULtiply Unsigned*) - mnożenie liczb bez znaku

Wynika to z faktu, że przy interpretacji liczb ujemnych jako uzupełnienia dwójkowego (U2), operacja mnożenia jest istotnie inna niż dla liczb bez znaku (co różni te operacje od dotad omówionych, gdzie funkcja arytmometru jest taka sama dla obu interpretacji). Ta sama uwaga odnosi się do operacji dzielenia.
W związku z tym, że wynik mnożenia dwóch liczb 16-bitowych jest 32-bitowy, w procesorze MC68000 dostępna jest tylko wersja operacji mnożenia argumentów 16-bitowych ($16 \times 16 \rightarrow 32$). Począwszy od MC68020 (w tym również dla CPU32) umożliwiono operację mnożenia danych długich (`.L`) w dwóch wariantach ($32 \times 32 \rightarrow 32$ i $32 \times 32 \rightarrow 64$).
Wersja podstawowa pozwala mnożyć dowolnie zaadresowany pierwszy argument przez mniej znaczące słowo rejestru danych:

```
                    D/A         Dn
                  -d(A,X)+      ---           X N Z V C
          MULS.W    abs    ,    ---           - ! ! ! 0
                  d(PC,X)       ---
                    imm         ---
```

Przykład takiego możenia:

```
MNOZNA      DS.W    1
MNOZNIK     DS.W    1
ILOCZYN     DS.L    1

            MOVE.W  MNOZNA,D0           (MNOZNA) -> D0.W
            MULS.W  MNOZNIK,D0          (MNOZNIK) * (D0.W) -> D0.L
            MOVE.L  D0,ILOCZYN          (D0.L) -> ILOCZYN
```

Analogicznie, lecz dla liczb bez znaku, działa instrukcja `MULU`.
Wersje rozszerzone (32/64-bitowe) wyglądają następująco:

```
            MULS.L  MNOZNIK,D0          (MNOZNIK) * (D0.L) -> D0.L
            MULS.L  MNOZNIK,D1-D0       (MNOZNIK) * (D0.L) -> D1.L:D0.L
```

Nadmiar może wystąpić tylko przy mnożeniu $32 \times 32 \rightarrow 32$.
Instrukcje dzielenia występują również w dwóch wariantach (`DIVS` i `DIVU`). W procesorze MC68000 dzielna jest 32-bitowa, dzielnik – 16-bitowy, a wynik składa się z ilorazu i reszty, które są umieszczane w rejestrze docelowym odpowiednio na dolnym i górnym słowie. Flagi są ustawiane na podstawie ilorazu.

```
                  D/A         Dn
                -d(A,X)+      ---          X N Z V C
DIVS.W            abs    ,    ---          - ! ! ! 0
                d(PC,X)       ---
                  imm         ---
```

Przykład dzielenia:

```
DZIELNA     DS.L    1
DZIELNIK    DS.W    1
ILORAZ      DS.W    1
RESZTA      DS.W    1

            MOVE.L  DZIELNA,D0          (DZIELNA) -> D0.L
            DIVS.W  DZIELNIK,D0
*                       (D0.L) % (DZIELNIK) : (D0.L) / (DZIELNIK) -> D0.L
            MOVE.W  D0,ILORAZ           (D0.W) -> ILORAZ
            SWAP    D0
            MOVE.W  D0,RESZTA           (D0.W) -> RESZTA
```

Analogicznie, lecz dla liczb bez znaku, działa instrukcja `DIVU`.
Począwszy od MC68020 (w tym również dla CPU32) umożliwiono operację dzielenia danych długich (`.L`) w trzech wariantach ($32//32 \rightarrow 32$, $64//32 \rightarrow 32:32$ i $32//32 \rightarrow 32:32$). Wersje te mają następującą postać:

```
            DIVS.L  DZIELNIK,D0         (D0.L) / (DZIELNIK) -> D0.L
            DIVS.L  DZIELNIK,D1:D0      (D1.L):(D0.L) / (DZIELNIK) -> D0.L
*                                       (D1.L):(D0.L) % (DZIELNIK) -> D1.L
            DIVSL.L DZIELNIK,D1:D0      (D0.L) / (DZIELNIK) -> D0.L
*                                       (D0.L) % (DZIELNIK) -> D1.L
```

Operacje na liczbach dziesiiętnych kodowanych binarnie operują na cyfrach dziesiętnych upakowanych po dwie w bajcie. Dopuszczalne tryby adresowania źródła i przeznaczenia (podobnie jak w przypadku arytmetyki binarnej z rozszerzeniem) obejmują dwa przypadki (`Dn,Dm` i `-(An),-(Am)`). Wynika to z podobnych przyczyn jak poprzednio. Operacja te są przewidziane do wykorzystania w wersji wielobajtowej, z użyciem bitu X rejestru CCR.

```
                                            X N Z V C
ABCD          Dn      ,     Dm              ! ? ! ? !


                                            X N Z V C
ABCD        -(An)     ,   -(Am)             ! ? ! ? !
```

Uwaga: flagi X i C są ustawiane, jeśli nastąpiło przeniesienie z wyższej cyfry BCD w bajcie, flagi N i V są nieokreślone.
Operacja `SBCD` jest analogiczna do `ABCD`, ale wykonuje odejmowanie. `NBCD` daje taki sam wynik jak odjęcie argumentu i bitu X z CCR od wartości "0" (czyli dziesiętne dopełnienie argumentu).
Przesunięcia arytmetyczne polegają na przesuwaniu zawartości rejestru danych lub komórki pamięci w taki sposób, by zachować zgodność z jej interpretacją jako liczby ze znakiem w kodzie U2.
Liczbę l przy pozycyjnym zapisie dwójkowym na N pozycjach (bitach) przedstawia się jako:

$$l = \sum_{i=0}^{N-1} b_i \cdot 2^i$$

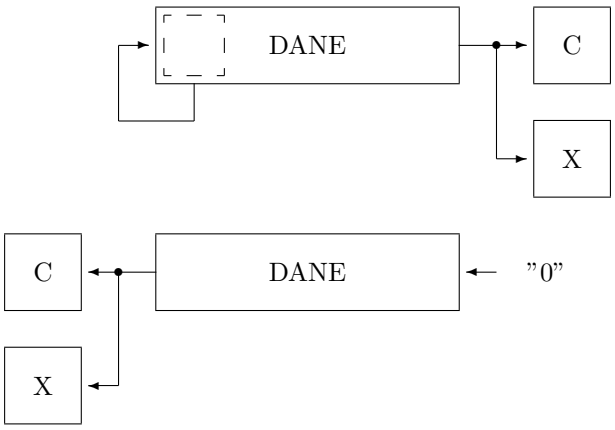
Po przesunięciu w lewo:

$$\sum_{i=1}^{N} b_{i-1} \cdot 2^i = \sum_{i=0}^{N-1} b_i \cdot 2^{i+1} = 2 \cdot \sum_{i=0}^{N-1} b_i \cdot 2^i = 2 \cdot l$$

Po przesunięciu w prawo:

$$\sum_{i=-1}^{N-2} b_{i+1} \cdot 2^i = \sum_{i=0}^{N-1} b_i \cdot 2^{i-1} = 2^{-1} \cdot \sum_{i=0}^{N-1} b_i \cdot 2^i = 2^{-1} \cdot l$$

Zwróćmy uwagę, że na ogół po przesunięciu liczba zajmuje o jedną pozycję bitową więcej, niż przed przesynięciem. przy przesynięciu o jedną pozycję, rolę bitu wydłużającego słowo w odpowiednią stronę pełni bit C w rejestrze CCR (dublowany przez bit X).
Ogólniej, przesunięcie liczby o n bitów w lewo ma spowodować pomnożenie, a w prawo – podzielnie liczby przez 2^n. W związku z tym przy przesunięciu w lewo flagi C i X w CCR są ustawiane na "1" gdy bit znaku był przed ostatnim przesunięciem równy "1", a flaga V jest ustawiana na "1" w przypadku nadmiaru, czyli wystąpienia zmiany bitu znaku w dowolnej chwili w trakcie operacji przesuwania.



Przy przesuwaniu w prawo do flag C i X w CCR wpisywany jest ostatni z wysuniętych z najmłodszej pozycji w liczbie bitów. Problem nadmiaru nie występuje, ponieważ (dla zachowania założonej wcześniej zgodnści z kodem U2) najbardziej znaczący bit (traktowany w U2 również jako bit znaku) jest kopiowany na swą pozycję. Dzięki temu nie ma możliwości zmiany znaku liczby, a więc wystąpienia nadmiaru.
W procesorach M68K mamy do wyboru dwa warianty przesunięcia arytmetycznego w zależności od miejsca docelowego. Jeśli operujemy na liczbie w rejestrze danych, możemy ją przesunąć w dowolnym z kierunków o zadaną ilość bitów, jeśli operujemy na komórce pamięci – zawsze przesuwamy tylko o jeden bit.
Wielkość przesunięcia w pierwszym przypadku może być zadana albo przez wartość natychmiastową, albo jako zawartość rejestru danych. Wartość natychmiastowa jest ograniczona do zakresu 1..8, ze względu na wielkość pola w kodzie rozkazu. Chcąć dokonać przesunięcia o długości większej od 8 lub zmiennej, musimy użyć innego rejestru danych, w którym umieszczamy wielkośc przesunięcia:

```
          Dm          Dn
    B     ---         ---          X N Z V C
ASL.W     ---    ,    ---          ! ! ! ! !
    L     ---         ---
          imm         ---

          Dm          Dn
    B     ---         ---          X N Z V C
ASR.W     ---    ,    ---          ! ! ! 0 !
    L     ---         ---
          imm         ---
```

Zgodnie z wcześniejszymi uwagami przy przesunięciu w prawo flaga nadmiaru jest zerowana.
Przesuwanie komórki pamięci wymaga adresu określającego miejsce docelowe,które musi być dostępne dla zapisu:

```
                          ---
    B                   -d(A,X)+        X N Z V C
ASL.W                     abs           ! ! ! ! !
    L                     ---
                          ---


                          ---
    B                   -d(A,X)+        X N Z V C
ASR.W                     abs           ! ! ! 0 !
    L                     ---
                          ---
```

Wielkość przesunięcia jest stała i wynosi 1 bit. Flagi zachowują się tak, jak omówiono wcześniej.
Przykładem zastosowania takich instrukcji może być mnożenie (lub dzielenie) liczb przez potęgi dwójki, ponieważ są one znacznie szybsze od `MUL` i `DIV`:

```
* podziel zawartosc D5.W przez 4:
           ASR.W   #2,D5
* pomnoz zawartosc D6.L przez 2&^(D5):
           ASL.L   D5,D6
* pomnoz zawartosc dlugiego slowa WYNIK przez 2:
           ASL.L   WYNIK
```

Arytmetyczne operacje pomocnicze to `EXT`, `NEG` i `NEGX`.
Instrukcja `EXT` (*sign EXTend*) służy do dopasowywania długości danych różnych typów, rozumianych jako liczby ze znakiem (U2) przed wykonywaniem na nich operacji arytmetycznych. Miejscem wykonania operacji jest dowolny z rejestrów danych. Można dokonywać rozszerzania do słowa 16-bitowego (`.W`) lub 32-bitowego (`.L`). W pierwszym przypadku (rozszerzenie bajtu do słowa) bit b_7 jest kopiowany na pozycje $b_8..b_{15}$, w drugim (rozszerzenie słowa do długiego słowa) bit b_{15} jest kopiowany na pozycje $b_{16}..b_{31}$.

```
                                        X N Z V C
 EXT.W                    Dn            - ! ! 0 0
    L
```

Zauważmy, że dla dopasowania bajtu do długiego słowa trzeba dwukrotnie wykonaćinstrukcję `EXT`:

```
* za/ladowanie bajtu do rejestru:
           MOVE.B  DANE,D3
* rozszerzenie bajtu do slowa:
           EXT.W   D3
* rozszerzenie slowa do dlugiego slowa:
           EXT.L   D3
```

W procesorach Motoroli, począwszy od 68020 (w tym w CPU32) wprowadzono dodatkową instrukcję `EXTB.L`, która zastępuje powyższą sekwencję jednym rozkazem:

```
* za/ladowanie bajtu do rejestru:
           MOVE.B  DANE,D3
* rozszerzenie bajtu do dlugiego slowa:
           EXTB.L  D3
```

Instrukcja `NEG` (*NEGate*) służy do znajdowania dopełnienia dwójkowego liczby zawartej w miejscu docelowym. W przyjętej konwencji U2 oznacza to zmianę znaku liczby na przeciwny. Dla liczby binarnej w N-bitowej reprezentacji:

$$NEG(l) = 0 - \sum_{i=0}^{N-1} b_i \cdot 2^i$$

gdzie

$$l = \sum_{i=0}^{N-1} b_i \cdot 2^i$$

było zadaną liczbą.
Dopuszczalne są wszystkie rozmiary argumentu, który musi być dostępny dla zapisu. Flagi są ustawiane w sposób naturalny, jak przy odejmowaniu argumentu od zera:

```
                    Dn
     B             -d(A,X)+        X N Z V C
NEG.W               abs             ! ! ! ! !
     L              ---
                    ---
```

Instrukcja `NEGX` (*NEGate with eXtend*) działa analogicznie, lecz dodatkowo uwzględnia flagę rozszerzenia (X w CCR) przy odejmowaniu od zera:

$$NEGX(l) = 0 - \sum_{i=0}^{N-1} b_i \cdot 2^i - (CCR : X)$$

```
                    Dn
     B             -d(A,X)+        X N Z V C
NEGX.W              abs             ! ! ! ! !
     L              ---
                    ---
```

2.3.4 Operacje logiczne

Instrukcje logiczne procesora M68K to:

- operacje logiczne (`AND`, `ANDI`, `OR`, `ORI`, `EOR`, `EORI`, `NOT`)
- przesunięcia i obroty logiczne (`LSL`, `LSR`, `ROL`, `ROXL`, `ROR`, `ROXR`)
- operacje bitowe (`BTST`, `BSET`, `BCLR`, `BCHG`, `TAS`)

Wszystkie operacje logiczne procesorów M68K (`AND`, `ANDI`, `OR`, `ORI`, `EOR`, `EORI`, `NOT`) polegają na wykonaniu odpowiednich funkcji logicznych osobno na każdym z bitów danych biorących udział w ich wykonaniu. W zależności od rozmiaru danych może to być 8, 16 lub 32 bity.
Instrukcje `AND`, `OR` i `EOR` (*Exclusive OR*) są adresowane bardzo podobnie do omawianych wcześniej 1.5-argumentowych instrukcji arytmetycznych:

```
                  D/A           Dn
  (OR) B       -d(A,X)+         ---          X N Z V C
   AND.W         abs      ,     ---          - ! ! 0 0
(EXOR) L       d(PC,X)          ---
                  imm           ---

                  Dn            Dm
  (OR) B          ---         -d(A,X}+       X N Z V C
   ADD.W          ---      ,     abs         - ! ! 0 0
 (EOR) L          ---           ---
                  ---           ---
```

Flagi znaku (N) i zerowego wyniku (Z) są ustawiane naturalnie, flaga rozszerzenia (X) nie jest zmieniana, a flagi nadmiaru (V) i przeniesienia (C) są zerowane.
Podobnie jak przy rozkazach arytmetycznych, istnieją wersje powyższych instrukcji dla danych natychmiastowych, umożliwiające użycie stałych argmentów bez zajmowania na nie dodatkowej pamięci:

```
                  ---         Dm
       (ORI) B    ---       -d(A,X}+        X N Z V C
        ANDI.W    ---    ,    abs           - ! ! 0 0
      (EORI) L    ---         ---
                  imm         ---
```

Co więcej, dopuszczalnym miejscem przeznaczenia dla 8-bitowego rozmiaru tych instrukcji jest rejestr flagowy (CCR), co pozwala operować na bbitach flagowych:

```
                  ---
         ORI.B    ---                         X N Z V C
        ANDI.B    ---    ,    CCR             ! ! ! ! !
        EORI.B    ---
                  imm
```

Typowe zastosowania instrucji logicznych polegają na operowaniu na fragmentach bajtów (słów, długich słów) np. przy modyfikowaniu zawartości pewnych pól w rejestrach portów:

```
* chcemy zmienic numer poziomu przerwania (bity 0..2) w rejestrze ICR
* na wartosc podana w rejestrze D0
           MOVE.B  ICR,D1              czytamy biezaca zawartosc ICR
           ANDI.B  #%11111000,D1       zerujemy trzy dolne bity
           ANDI.B  #%00000111,D0       wybieramy wlasciwa czesc D0
           OR.B    D0,D1               sk/ladamy bajt z czesci
           MOVE.B  D1,ICR              zapisujemy nowa zawartosc ICR
```

W przypadku CCR możemy zerować lub ustawiać na "1" dowolne flagi:

```
* zerowanie C (przeniesienia):
           ANDI.B  #%11110,CCR         flaga C - bit b0 w CCR
* ustawianie X i V:
           ORI.B   #%10010,CCR         X - bit b4, V - bit b1 w CCR
* zmiana X i C na przeciwne:
           EORI.B  #%10001,CCR
```

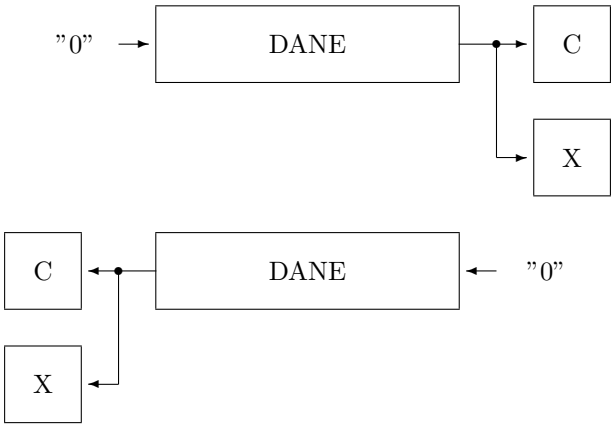
Instrukcja `NOT` stanowi odpowiednik `NEG` dla instrukcji logicznych. Jej wykonanie powoduje zmianę wartości każdego bitu danych zawartych w miejscu docelowym wskazywanym przez jedyny argument na przeciwną. Dostępne są wszystkie rozmiary danych, a flagi są obsługiwane analogicznie jak przy wcześniej omawianych rozkazach logicznych:

```
                            Dn
            B              -d(A,X)+        X N Z V C
         NOT.W               abs           - ! ! 0 0
            L                ---
                             ---
```

Instrukcje przesunięć logicznych (`LSL`, `LSR`) mają takie same formy adresowania jak przesunięcia arytmetyczne:

```
                  Dm          Dn
            B     ---         ---           X N Z V C
      (LSL) LSR.W ---    ,    ---           ! ! ! 0 !
            L     ---         ---
                  imm         ---


                              ---
            B                -d(A,X)+       X N Z V C
      (LSL) LSR.W              abs           ! ! ! 0 !
            L                  ---
                               ---
```

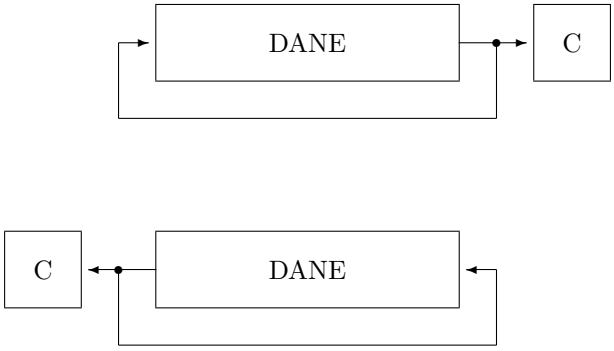
Flaga nadmiaru jest zerowana jak przy wcześniej omawianych operacjach logicznych, a flagi C i X są ustawiane zgodnie z wartością bitu wysuniętego poza obszar danych.



Istotna różnica występuje przy przesunięciu w prawo. Zamiast powielania najstarszego bitu w słowie (bitu znaku przy reprezentacji U2) odpowiedni bit wyniku jest **zerowany**. W ten sposób mamy pełną symetrię pomiędzy `LSL` i `LSR`, które tym razem realizują odpowiednio mnożenie i dzielenie liczb (tym razem bez znaku) przez potęgę liczby 2.
Instrukcje obrotów logicznych występują w dwóch odmianach: zwykłej i rozszerzonej. Argumenty wszystkich wersji mają takie samo znaczenie jak w przesunięciach, ale różni się operowanie na flagach. W przypadku obrotów zwykłych (`ROL` – *ROtate Left*, `ROR` – *ROtate Right*) flaga X jest nieużywana:

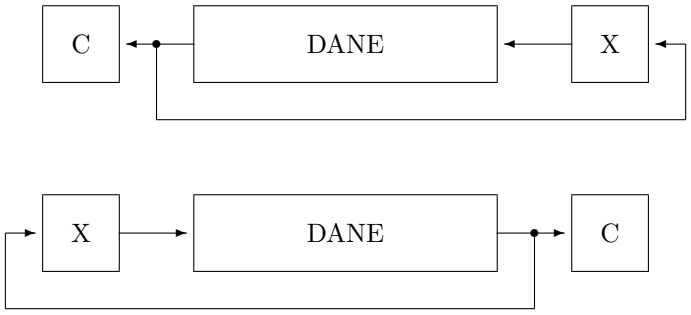
```
                  Dm          Dn
          B       ---         ---           X N Z V C
(ROL) ROR.W       ---    ,    ---           - ! ! 0 !
          L       ---         ---
                  imm         ---

                              ---
          B                 -d(A,X)+        X N Z V C
(ROL) ROR.W                   abs           - ! ! 0 !
          L                   ---
                              ---
```

Wykonywana jest podobna operacja, jak przy odpowiednim przesunięciu logicznym, ale wysunięty z pola danych bit trafia do flagi C i na zwolnione w czasie przesunięcia miejsce na przeciwległym krańcu pola danych.



Przy wersjach rozszerzonych (`ROXL` – *ROtate with eXtend Left*, `ROXR` – *ROtate with eXtend Right*) flaga X jest modyfikowana. Obrót jest wykonywany na danych wydłużonych o bit flagowy X. Wysuięty z pola danych bit trafia do flag C i X, a na zwolnione przy przesunięciu miejsce trafia dotychczasowa zawartość flagi X.

```
                    Dm          Dn
              B     ---         ---           X N Z V C
(ROXL) ROXR.W       ---    ,    ---           ! ! ! 0 !
              L     ---         ---
                    imm         ---

                                ---
              B                -d(A,X)+       X N Z V C
(ROXL) ROXR.W                   abs           ! ! ! 0 !
              L                 ---
                                ---
```



Dzięki temu można dokonywać przesunięć logicznych danych dłuższych niż 32-bitowe:

```
* dane do przesuniecia w lewo o 1 bit:
DANE        DS.L    3

* przesuniecie logiczne w lewo (od najmlodszego slowa):
            LSL.L   DANE+2*4            przesun trzecie slowo danych
            ROL.L   DANE+1*4            obroc drugie slowo danych
            ROL.L   DANE                obroc pierwsze slowo danych
```

Przesuwanie długich liczb w prawo rozpoczynamy od najstarszego słowa:

```
* dane (bez znaku) do przesuniecia w prawo o 1 bit:
DANE        DS.L    3

* przesuniecie logiczne w prawo (od najstarszego slowa):
            LSR.L   DANE                przesun pierwsze slowo danych
            ROR.L   DANE+1*4            obroc drugie slowo danych
            ROR.L   DANE+2*4            obroc trzecie slowo danych
```

Zwróćmy uwagę, że analogiczna operacja musi być wykonana dla liczb ze znakiem (w kodzie U2), ale pierwsza instrukcja jest wtedy przesunięciem arytmetycznym:

```
* dane (ze znakiem - U2) do przesuniecia w prawo o 1 bit:
DANE        DS.L    3

* przesuniecie arytmetyczne w prawo (od najstarszego slowa):
            ASR.L   DANE                przesun pierwsze slowo danych
            ROR.L   DANE+1*4            obroc drugie slowo danych
            ROR.L   DANE+2*4            obroc trzecie slowo danych
```

Dostęp do pojedynczych bitów, zarówno w 32-bitowych rejestrach danych procesora, jak i w 8-bitowych bajtach pamięci umożliwiaą instrukcje bitowe (`BTST` – *Bit TeST*, `BSET` – *Bit test and SET*, `BCLR` – *Bit test and CLeaR*, `BCHG` – *Bit test and CHanGe*). Adres pojedynczego bitu składa się z wynikowego adresu miejsca docelowego i numeru bitu. W przymadku, gdy miejscem docelowym jest rejestr danych, numer bitu może wynosić od 0 do 31. W pamięci można operować tylko na bajtach, więc numer bitu może wynosić od 0 do 7. Argument określający numer bitu jest brany modulo 32 lub modulo 8 (odpowiednio do miejsca docelowego).
Zaczniemy od instrukcji `BTST` ustawiającej flagę Z w zależności od stanu zaadresowanego bitu:

```
                Dn          ---
                ---       -d(A,X}+        X N Z V C
BTST.B          ---    ,    abs           - - ! - -
                ---       d(PC,X)
                imm         ---

                Dn          Dm
                ---         ---           X N Z V C
BTST.L          ---    ,    ---           - - ! - -
                ---         ---
                imm         ---
```

Jak wspomniano wcześniej, numer bitu może się znajdować w rejestrze danych (rozkaz zajmuje 1 słowo kodu) lub być podany jako dana natychmiastowa (2 słowa kodu). Jest on brany modulo 8 (`.B`) lub modulo 32 (`.L`).
Trzy pozostałe instrukcje z omawianej grupy ustawiają flagę Z w identyczny sposób, ale dodatkowo, **po dokonaniu testu**, modyfikują wskazany bit. `BCLR` zeruje, `BCHG` neguje, a `BSET` ustawia na "1" bit zaadresowany przez podanie adresu wynikowego i numeru bitu. Z dopuszczalnych trybów adresowania pamięci trzeba więc wyłączyć adresowanie względem PC:

```
                Dn          ---
(BCLR)          ---       -d(A,X}+        X N Z V C
  BCHG.B        ---    ,    abs           - - ! - -
(BSET)          ---         ---
                imm         ---

                Dn          Dm
(BCLR)          ---         ---           X N Z V C
  BCHG.L        ---    ,    ---           - - ! - -
(BSET)          ---         ---
                imm         ---
```

Opisane instrukcje pozwalają badać i ustawiać bity w jednym rozkazie maszynowym, co znacznie przyśpiesza wykonywanie programów. Jeśli bit w ślowie wykorzystamy jako tzw. semafor, czyli znacznik dostępności zasobu, to stan "1" może oznaczać "zajęty", a stan "0" – "wolny". Wtedy moglibyśmy użyć w procesie żądającym zasobu instrukcji `BSET` i (w zależności od jej wyniku – flagi Z w CCR) czekać na zwolnienie, lub rozpocząć korzystanie z zasobu. Zauważmy, że instrukcja `BSET` po sprawdzeniu stanu semafora ustawiła go na "1". Jeśli był ustawiony wcześniej, to nic sięńie zmieniło, a proces musi czekać, jeśli był równy "0", to właśnie został ustawiony na "1", a proces może korzystać z zasobu.
Można jednak sobie wyobrazić sytuację, że nie wystarczy to do bezpiecznego dostępu do wspólnych zasobów przez dwa procesy działające na różnych procesorach. Wynika to z faktu, że omówione rozkazy dostają się

do lokacji w pamięci w celu odczytania bajtu i jego zapisania po zmodyfikowaniu w dwóch osobnych cyklach magistrali. Przy pracy wieloprocesorowej, gdy kilka jednostek ma dostęp do bajtu używanego jako semafor, możliwe jest rozdzielenie tych dostępów przez dosstęp innego procesora. Jeśli ten dostęp miał na celu właśnie sprawdzenie zajętości zasobu strzeżonego przez omawiany semafor, to **oba procesory** dostrzegą, że zasób jest wolny i przystąpią do jego wykorzystywania, co spowoduje błędy.
Aby tego uniknąć trzeba mieć do dyspozycji instrukcję, w której odczyt i zapis bajtu zawierającego semafor nie mogą zostać rozdzielone. Taką instrukcją jest `TAS` (*Test And Set*):

	Dn	
	-d(A,X}+	X N Z V C
BTST.B	abs	- ! ! 0 0

Sprawdza ona stan bajtu wskazanego przez adres wynikowy i odpowiednio ustawia flagi N i Z w CCR, zeruje flagi V i C, oraz ustawia na "1" bit b_7 testowanego bajtu. Dostęp jest wykonywany w jednym, niepodzielnym cyklu (*read–modify–write*), co spełnia warunki bezpiecznego dzielenia zasobów omówione wcześniej.

2.3.5 Instrukcje sterowania przebiegiem programu

Dotychczas omawiane instrukcje operowały na danych. Do konstruowania programów potrzebne są oprócz nich instrukcje sterujące, pozwalające na tworzenie pętli różnych typów, wykonywanie skoków, wywoływanie procedur. w ich skład wchodzą:

- skoki (warunkowe i bezwarunkowe): `JMP`, `BRA`, `Bcc`
- warunkowe ustawianie flag: `Scc`
- pętle warunkowe: `DBcc`
- instrukcje porównywania: `TST`, `CMP`, `CMPA`, `CMPI`, `CMPM`
- obsługa procedur `JSR`, `BSR`, `RTS`, `RTR`, `LINK`, `UNLK`
- przerwania programowe `TRAP`, `TRAPV`
- instrukcje pomocnicze `NOP`, `ILLEGAL`, `CHK`

Najczęstszym sposobem modyfikowania licznika rozkazów jest jego automatyczne zwiększanie w miarę pobierania kolejnych słów kodu. Zmiana zawartości PC na podany adres (spod którego ma być pobrany kod następnej instrukcji) może być wykonana w sposób bezwarunkowy przy pomocy instrukcji `JMP` (*JuMP*):

```
                    ---
                  d(A,X)          X N Z V C
JMP                 abs           - - - - -
                  d(PC,X)
                    ---
```

Dopuszczalne tryby adresowania muszą się odwoływać do miejsca w przestrzeni adresowej pamięci. Żadne flagi nie są modyfikowane.
Zwróćmy uwagę, że ten rozkaz pozwala, w zależności od użytego trybu adresowania, uzyskać kod przesuwalny lub absolutny. Czasem potrzebny jest dostęp do procedur wbudowanych w system operacyjny lub monitor rezydujący w pamięci stałej. Wtedy można skorzystać z absolutnego trybu adresowania by uruchomić odpowiedni fragment kodu. Nie jest to jednak rozwiązanie godne polecenia. Zastosowanie adresowania względem rejestru adresowego pozwala (w miarę potrzeby) uzyskać odwołanie bezwzględne lub względne, w zależności od sposobu zainicjowania tego rejestru (por. `LEA`, `MOVEA #`). Adresowanie względem licznika rozkazów pozwala uzyskać kod przesuwalny (*PIC – Position Independent Code*) bez dodatkowych zabiegów. Dodatkowo, przy trybach indeksowych można łatwo skonstruować tablicę skoków realizującą rozgałęzienie programu.
Najczęstszym sposobem adresowania potrzebnym do realizacji lokalnych pętli jest adresowanie względem licznika rozkazów. Omówiony rozkaz `JMP` w trybie `d(PC)` może służyć do tego celu. Oprócz niego istnieje jednak również cała rodzina specjalnych rozkazów skoków względnych (bezwarunkowych i warunkowych), wywodzących się z wcześniejszych mikroprocesorów Motoroli.
Skok względny bezwarunkowy zie się `BRA` (*BRanch Always*). Jego rgumentem w symbolicznym zapisie asemblerowym jest etykieta określająca miejsce w tekście źródłowym, od którego należy kontynuować wykonywanie programu:

```
            BRA     ET1
* tu: kod do ominiecia
            ...
* tu nalezy kontynuowac ET1
            ...
```

W kodzie binarnym tego rozkazu umieszczona jest wartość przesunięcia pomiędzy pozycją rozkazu `BRA` (dokładniej: pozycją licznika rozkazów **po pobraniu** słowa z kodem maszynowym `BRA`), a pozycją etykiety stanowiącej argument w programie źródłowym (`ET1`). Wielkość ta, jako liczba ze znakiem w kodzie U2, może się mieścić na jednym lub dwóch bajtach. W przypadku małych odległości względnych (lokalne skoki) jeden bajt jest wbudowany do słowa kodowego:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   1   1   0   0   0   0   0  '--------------v--------------'
                                                 8-bitowe przesuniecie
```

Jeśli odległość jest większa, przesunięcie w kodzie jest ustawiane na 0, a kolejne słowo kodu zawiera 16-bitowe przesunięcie (U2). Możliwe jest również przesunięcie 32-bitowe (pole w kodzie ustawione na $FF), ale dopiero dla procesorów od MC68020 wzwyż.
Wybór odpowiedniego rozmiaru przesunięcia następuje podczas tłumaczenia postaci źródłowej na binarną. Jeśli asembler zna wartość etykiety docelowej **przed** tłumaczeniem rozkazu skoku względnego, to może określić wielkość przesunięcia, a więc i rozmiar argumentu. Jeśli etykieta docelowa nie jest znana przy pierwszym przebiegu asemblera (jak w powyższym przykładzie), to przyjmuje się gorszy przypadek (16-bitowe przesunięcie) i przydziela dwa słowa na kod rozkazu `BRA`. Jak widać, skoki wstecz są zawsze tłumaczone optymalnie. Ponieważ zazwyczj pogramista potrafi ocenić odległość skoku naprzód, wprowadzono konwencję pozwalającą zmusić asembler do przydzielenia jednego słowa kodu "na odpowiedzialność programisty". Używa się w tym celu znacznika `.S` (*Short*) po nazwie skoku względnego (w naszym przykładzie: `BRA.S`). Jeśli po wyliczeniu wartości etykiet (w drugim przebiegu asemblera) okazałoby się, że przesunięcie jest większe, niż można wyrazić 8-bitową liczbą ze znakiem (przy skoku naprzód powyżej 127), to zostanie zasygnalizowany błąd asemblacji. Analogiczne (`.L`) można wymusić długą formę adresowania.
Skoki względne tworzą rodzinę 16 rozkazów, która operuje omówionym wyżej sposobem adresowania, a różnią się warunkami, przy których spełnieniu następuje zmiana zawartości licznika rozkazów (i w konsekwencji – zmiana sekwencyjnego wykonywania instrukcji programu). `BRA` jest szczególnym przypadkiem takiego rozkazu. Ogólnie oznacza się te skoki nazwami `Bcc`, gdzie `cc` stanowi dwuliterowy skrót warunku (*Condition Code*).

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   1   1   0  '------v------' '--------------v--------------'
                          kod warunku        8-bitowe przesuniecie
```

Warunki są określone przez funkcje logiczne kombinacji bitów flagowych CCR (*Condition Code Register*). Kod warunku jest umieszczony w słowie rozkazowym. Poszczególne warunki są zestawione w tabeli.

kod	symbol	nazwa warunku	funkcja logiczna
0000	T RA	True Always	1
0001	F NE	False NEver	0
0010	HI	HIgher	$\overline{C} \cdot \overline{Z}$
0011	LS	Lower or Same	$C + Z$
0100	CC HS	Carry Clear Higher or Same	$\overline{C}$
0101	CS LO	Carry Set LOwer	C
0110	NE	Not Equal	$\overline{Z}$
0111	EQ	EQual	Z
1000	VC	oVerflow Clear	$\overline{V}$
1001	VS	oVerflow Set	V
1010	PL	PLus	$\overline{N}$
1011	MI	MInus	N
1100	GE	Greater or Equal	$N \cdot V + \overline{N} \cdot \overline{V}$
1101	LT	Less Than	$N \cdot \overline{V} + \overline{N} \cdot V$
1110	GT	Greater Than	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
1111	LE	Less or Equal	$Z + N \cdot \overline{V} + \overline{N} \cdot V$

Wymienione warunki zastosowane do skoków mogą służyć do uzależnienia przebiegu programu od wyników przetwarzania danych:

```
            MOVE    WYNIK,D0        pobierz wynik
            SUB     PROG,D0         odejmij prog
            BGT     PONAD           skocz, jesli wynik > prog
* tu gdy wynik <= prog
PONIZEJ     ...
            ...
* tu gdy wynik > prog
PONAD       ...
```

Ponieważ czasem zachodzi potrzeba zapamiętania złożonego warunku opartego na powyższej tabeli do późniejszego wykorzystania, wprowadzono rodzinę pomocniczych rozkazów `Scc` ustawiających bajt zaadresowany przez argument na wartość logiczną odpowiadającą warunkowi. Prawda (*TRUE*) jest reprezentowana przez %11111111, a fałsz (*FALSE*) – przez %00000000. Flagi w CCR nie są modyfikowane, miejsce docelowe musi być zapisywalne jako bajt:

```
                                  Dn
                               +d(A,X)-          X N Z V C
             Scc                  abs            - - - - -
                                  ---
                                  ---
```

W poprzednim przykładzie moglibyśmy więc wprowadzić własną flagę oznaczającą przekroczenie progu przez wynik:

```
            MOVE    WYNIK,D0        pobierz wynik
            SUB     PROG,D0         odejmij prog
            SGT     NAD_PROG        ustaw flage, jesli wynik > prog
* tu dalsza czesc kodu
            ...
* dopiero tu testujemy nasza flage
            TST     NAD_PROG        sprawdz flage
            BNE     PONAD           skocz, jesli flaga ustawiona
* tu gdy wynik <= prog
PONIZEJ     ...
            ...
* tu gdy wynik > prog
PONAD       ...
```

Przy budowie pętli iteracyjnych najczęściej spotyka się zmniejszanie (lub zwiększanie) licnika obiegów o 1. Dodatkowo bywa testowany warunek zakończenia pętli nie mający związku z licznikiem. Trzy działania potrzebne do realizacji takiej pętli skupiono w grupie instrukcji pętlowych `DBcc` (*Decrement and Branch until Condition Code*).
Pierwszym argumentem instrukcji jest mniej znaczące słowo rejestru danych stanowiące licznik (zmniejszany o 1 przy każdym obiegu). Drugim argumentem jest etykieta miejsca, do którego należy skoczyć, o ile warunek (`cc`) **nie jest spełniony** i licznik nie stał się równy −1:

```
PETLA       ...                         repeat{ ...
            ...                             ...
            Dcc     D0,PETLA            } until( cc || --D0==-1 );
```

Rozważmy przykład dla warunku końcowego, który nigdy nie jest spełniony (FALSE):

```
            MOVE    ILOSC,D0            d0=ilosc;
            SUBQ    #1,D0               d0--;
* tu poczatek petli                     repeat{
PETLA       ...                             ...
            ...                             ...
            DBF     D0,PETLA            } until(FALSE || --d0==-1);
```

Jak widać, jest to zwykła pętla iteracyjna:

```
    for(d0=ilosc-1;d0!=-1;--d0){
        ...
    }
```

Przykład pętli z nietrywialnym warunkiem końcowym:

```
Napis       DS.B    Dlugosc             bufor na napis
*                                       koniec napisu -- bajt==0
            LEA.L   Napis(PC),A0        inicjacja rejestru bazowego
            MOVE    #Dlugosc-1,D0       inicjacja licznika

Loop        MOVE.B  (A0)+,Port          wysylanie kolejnych znakow napisu
            DBEQ    D0,Loop             ...do bajtu==0, lub konca bufora
```

demonstruje zwartość kodu uzyskanego dla dość złożonego zadania (konieczność sprawdzania dwóch warunkow końcowych):

```
    char napis[dlugosc];

    a0=&napis;
    d0=dlugosc-1;

    repeat{
    } until( (0==(port=*a0++)) || (--d0==-1) );
```

Zauważmy, że testując zawartość licznika pętli po jej zakończeniu możemy stwierdzić, który z warunków zakończenia został wcześniej spełniony. Jeśli licznik (w przykładach D0) jest równy -1, to pętla zakończyła się na skutek wyczerpania licznika, w przeciwnym razie – zadziałał warunek `cc` w instrukcji `DBcc`.
Jak było widać wcześniej, przy omawianiu różnych instrukcji, flagi w rejestrze CCR nie zawsze są ustawiane w sposób naturalny, zgodny z ich nazwami. Mnemotechniczne nazwy większości warunków podanych w tabeli są odpowiednie, jeśli skok (`Bcc`), ustawianie flagi (`Scc`), lub zamknięcie pętli (`DBcc`) następuje **bezpośrednio** po instrukcji arytmetycznej lub logicznej. Aby umożliwić stosowanie operacji warunkowych w innych sytuacjach, wprowadzono instrukcje testowania i porównywania danych ustawiające flagi w sposób naturalny, a więc zgodny z interpretacją nazw warunków.
Jednoargumentowa instrukcja `TST` ustawia tylko flagi statyczne (podonie jak `MOVE`).

```
                    Dn
      B            -d(A,X)+        X N Z V C
TST.W               abs             - ! ! 0 0
      L             ---
                    ---
```

Jej użycie zilustrowano wcześniej (przy okazji `Scc`).
Instrukcje porównania są 1.5-argumentowe, podobnie jak instrukcje arytmetyczne. Szczególne podobieństwo łączy je z instrukcjami odejmowania. Flagi są ustawiane odpowiednio do wyniku podobnej operacji odejmowania argumentu źródłowego od docelowego, ale **argument docelowy nie jest modyfikowany**. Dodatkowo, flaga rozszerzenia (X) **nie jest modyfikowana**.
Instrukcja `CMP` (*CoMPare*) służy do porównywania danych o rozmiarze od 8- do 32-bitowych słów, z zawartością rejestru danych:

```
                D/A          Dn
     B        -d(A,X)+       ---           X N Z V C
CMP.W           abs     ,     ---           - ! ! ! !
     L        d(PC,X)        ---
                imm          ---
```

Flagi są ustawiane jak po operacji:

$$(D_n) - < source >$$

Do porównywania adresów przeznaczona jest instrukcja `CMPA` (*CoMPare Address*). pozwala ona porównywać argumenty 16- i 32-bitowe z zawartością rejestru adresowego, ale zawsze traktuje je jako 32-bitowe liczby z ewentualnym rozszerzeniem znaku.

```
                D/A          An
     B        -d(A,X)+       ---           X N Z V C
CMPA.W          abs     ,     ---           - ! ! ! !
     L        d(PC,X)        ---
                imm          ---
```

Flagi są ustawiane jak po operacji:

$$(A_n) - < source >$$

Istnieje wersja operacji porównywania z argumentem natychmiastowym (`CMPI` - *CoMPare Immediate*), działająca podobnie do `SUBI`:

```
                ---          Dm
     B          ---       -d(A,X}+         X N Z V C
CMPI.W          ---     ,    abs            - ! ! ! !
     L          ---          ---
                imm          ---
```

Można ją wykorzystać do sprawdzenia, czy np. znak jest literą:

```
        CMPI    #'A',ZNAK
        BLO     NieLitera
        CMPI    #'Z',ZNAK
        BLS     Litera
        CMPI    #'a',ZNAK
        BLO     NieLitera
        CMPI    #'z',ZNAK
        BHI     NieLitera
* tu obsluga liter
Litera  ...

* tu obsluga innych znakow
NieLitera   ...
```

Dodatkowym ułatwieniem przy porównywaniu dwóch tablic w pamięci (np. napisów) jest instrukcja `CMPM` (*CoMPare Memory*). Ze względu na swoje przeznaczenie ma tylko jeden tryb adresowania, pośrdni z autoinkrementacją:

```
     B                                    X N Z V C
CMPM.W       (An)+    ,    (Am)+          - ! ! ! !
     L
```

Procedura porównywania dwóch napisów może wyglądać następująco:

```
Napis1      DS.B        Dlugosc
Napis2      DS.B        Dlugosc

            LEA.L       Napis1(PC),A1
            LEA.L       Napis2(PC),A2
            MOVE.W      #Dlugosc-1,D0       ustawienie licznika

Petla       CMPM.B      (A1)+,(A2)+         porownanie znakow
            DBNE        D0,Petla            ...do roznicy, lub konca tablic

            CMP.W       #-1,D0              czy koniec tablicy?
            BNE         NieRowne            nie, by/ly roznae znaki
* tu obsluga rownych napisow
Rowne       ...

* tu obsluga rownych napisow
NieRowne    ...
```

Do obsługi procedur służą instrukcje: `JSR`, `BSR`, `RTS`, `RTR`, `LINK`, `UNLK`. Dwie pierwsze stanowią wywołania podprogramów, dwie następne ułatwiają powrót do sekwencyjnego wykonywania programu, dwie ostatnie służą do obsługi lokalnych zmiennych używanych w podprogramie.
Instrukcja `JSR` (*Jump to SubRoutine*) stanowi skok bezwzględny do podprogramu. Adres podprogramu może być zadany w różnych trybach:

```
                                  ---
                                d(A,X)          X N Z V C
          JSR                     abs           - - - - -
                                d(PC,X)
                                  ---
```

Tryby pośrednie względem licznika rozkazów pozwalają uzyskać kod niezależny od położenia. Krótszą wersją względnie adresowanego wywołania procedury jest `BSR` (*Branch to SubRoutine*). Jest on (pod względem adresowania) identyczny z omówionym wcześniej skokiem względnym `BRA`:

```
bit:     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
          0   1   1   0   0   0   0   1  '--------------v--------------'
                                                8-bitowe przesuniecie
```

Odnoszą się też do niego wszystkie uwagi dotyczące większych przesunięć i składni asemblerowej.
Oba rozkazy wywołania procedur są wykonywane w identyczny sposób:

- wskaźnik stosu jest zmniejszany o 4: $(SP) - 4 \rightarrow SP$
- na szczyt stosu składowany jest licznik rozkazów: $(PC) \rightarrow (SP)$
- wynikowy adres skoku jest wpisywany do licznika rozkazów: $< ea > \rightarrow PC$

W ten sposób na szczycie stosu pojawia się adres, pod który należy powrócić po zakończeniu wywoływanej procedury. Instrukcja `RTS` (*ReTurn from Subroutine*) wykorzystuje adres powrotu znajdujący się na szczycie stosu (po którymś z opisanych wywołań procedury) do odtwozenia zawartości licznika rozkazów sprzed wywołania. Jest to adres następnej instrukcji po `BSR` lub `JSR`, ponieważ na stos trafia zawartość PC już **po pobraniu całego rozkazu skoku**:

- zawartość szczytu stosu jest wpisywana do licznika rozkazów: $((SP)) \rightarrow PC$
- wskaźnik stosu jest zwiększany o 4: $(SP) + 4 \rightarrow SP$

Rozkaz ten jest bezargumentowy i nie narusza żadnych flag w CCR.

```
            BSR     PROC
* tu chcemy wrocic po wykonaniu procedury
            ...

            ...
* procedura
PROC        ...
            ...
            RTS                         powrot z procedury
```

Instrukcja `RTR` dodatkowo odtwarza ze stosu zawartość rejestru flagoego (CCR):

- zawartość nieparzystego bajtu ze szczytu stosu jest wpisywana do rejestru flagowego: $((SP)+1) \rightarrow CCR$
- wskaźnik stosu jest zwiększany o 2: $(SP)+2 \rightarrow SP$
- zawartość szczytu stosu jest wpisywana do licznika rozkazów: $((SP)) \rightarrow PC$
- wskaźnik stosu jest zwiększany o 4: $(SP)+4 \rightarrow SP$

Jak widać, rozkaz ten umożliwia powrót z procedury z odpowiednio ustawionymi (lub zachowanymi) bitami flagowymi. Trzeba jednak pamiętać, że odpowiedzialność za składowanie CCR na stosie ciąży na programiście, który musi odpowiednią instrukcję umieścić w kodzie procedury:

```
            BSR     POMOC
* tu chcemy wrocic bez straty ustawienia flag
            ...

            ...
* procedura pomocnicza zachowujaca flagi
POMOC       MOVE    CCR,-(SP)           zachowanie flag na stosie
            ...
            RTR                         przywrocenie flag i powrot
```

Niektóre procedury mogą się obejść bez zmiennych lokalnych umieszczonych w pamięci. Duża ilość uniwersalnych rejestrów pozwala programiście umieścić w nich zmienne użuwane wewnątrz procedury, a na początku i na końcu procedury zadbać o zachowanie ich wcześniejszej zawartości i jej przywrócenie. Jest to szczególnie łatwe dzięki opisanej wcześniej instrukcji `MOVEM` adresowanej z automodyfikacją:

```
            BSR     PROC
* tu chcemy wrocic po wykonaniu procedury
            ...

            ...
* procedura uzywajaca niektorych rejestrow
PROC        MOVEM   D1-D4/A2-A4,-(SP)   zachowanie rejestrow
            ...
            MOVEM   (SP)+,D1-D4/A2-A4   odtworzenie rejestrow
            RTS                         powrot z procedury
```

Jeśli nieodzowne jest użycie większej ilości zmiennych lokalnych, pomocna jest para instrukcji `LINK`, `UNLK` (*UNLinK*).

Instrukcja `LINK` wymaga dwóch argumentów:

```
                                                    X N Z V C
        LINK          An     ,    imm               - - - - -
```

Pierwszy z argumentów jest rejestrem adresowym, który zostanie wykorzystany jako bieżący wskaźnik zarezerwowanej na zmienne ramki stosu, a drugi (dana natychmiastowa) określa wielkość rezerwowanego obszaru (w bajtach, liczba ze znakiem). Wykonanie tej instrukcji przebiega następująco:

- wskaźnik stosu jest zmniejszany o 4: $(SP) - 4 \rightarrow SP$
- na szczyt stosu składowany jest wybrany rejestr: $(An) \rightarrow (SP)$
- rejestr ten jest inicjowany jako wskaźnik początku obszaru danych: $(SP) \rightarrow An$
- wskaźnik stosu jest obniżany o zadaną wielkość obszaru: $(SP) + < imm > \rightarrow SP$

Operacja odwrotna (`UNLK`) wymaga tylko podania rejestru adresowego, który zawiera wskaźnik ramki stosu:

```
                                                    X N Z V C
        UNLK                  An                    - - - - -
```

Wykonanie tej instrukcji przebiega następująco:

- wskaźnik stosu jest odtwarzany z rejestru wskaźnikowego: $(An) \rightarrow SP$
- dawna zawartość rejestru wskaźnikowego jest odtwarzana ze szczytu stosu: $((SP)) \rightarrow An$
- wskaźnik stosu jest zwiększany o 4: $(SP) + 4 \rightarrow SP$

Obszar zarezerwowany na zmienne lokalne może być łatwo adresowany przy pomocy trybów adresowania względem rejestru wskaźnikowego:

```
            BSR     PROC
* tu chcemy wrocic po wykonaniu procedury
            ...

            ...
* procedura uzywajaca zmiennych lokalnych na stosie
PROC        LINK    A6,#-8*2            zachowanie miejsca na 8 slow
            ...
            MOVE.B  D1,-4(A6)           dostep do jednego ze slow
            ...
            UNLK    A6                  likwidacja ramki stosu
            RTS                         powrot z procedury
```

Łatwo zauważyć, że otrzymany w ten sposób kod jest wielodostępny i umożliwia tworzenie tzw. czystych procedur mogących pracować rekurencyjnie.
Zauważmy dodatkowo, że utworzona na stosie struktura jest listą, powiązaną przez kolejne wcielenia rejestru wskaźnikowego. Utrzymując konsekwentnie powyższą konwencję rozpoczynania procedur (używając `LINK An,#0` dla procedur bez zmiennych lokalnych) umożliwiamy programom uruchomieniowym automatyczne śledzenie zagłębienia wywołań procedur przez analizę wspomnianej listy na stosie (wskaźnikiem jej początku jest aktualna zawartość rejestru wskaźnikowego An).
Przerwania programowe `TRAP` służą przede wszystkim do odwoływania się do usług systemu operacyjnego, a w ogólności do obsługi operacji wymagających stanu uprzywilejowanego procesora (*supervisor state*). Nie są to instrukcje uprzywilejowane, lecz ich wykonanie wprowadza procesor w stan uprzywilejowany (systemowy). Składnia asemblerowa:

```
            TRAP    #n
```

gdzie `n` jest liczbą od 0 do 15, odpowiada 16 kodom instrukcji powodującym 16 różnych, choć równouprawnionych przerwań programowych. Obsługa przerwania programowego polega na:

- przełączeniu w tryb *supervisor*: $1 \rightarrow SR : S$
- wyskładowaniu PC i SR na stosie wskazywanym przez SSP (*supervisor Stack Pointer*): $(SSP) - 4 \rightarrow SSP$; $(PC) \rightarrow (SSP)$; $(SSP) - 2 \rightarrow SSP$; $(SR) \rightarrow (SSP)$
- załadowaniu do PC adresu obsługi trapu: $(\$80 + n * 4) \rightarrow PC$

Wyrażenie: $(\$80 + n * 4)$ oznacza umowny adres miejsca, gdzie musi się znajdować adres procedury obsługi przerwania programowego o numerze `n` (`TRAP #n`). Miejsca takie, przeznaczone na adresy obsługi różnych zdarzeń specjalnych (wektory) są umieszczone w najniższym kilobajcie przestrzeni adresowej procesora MC68000. W procesorach następnych (od 68010) wprowadzono dodatkowy rejestr VBR (*Vector Base Register*), który umożliwia szybką zmianę wszystkich wektorów dzięki temu, że wtedy opisywane wyrażenia ma postać: $((VBR) + \$80 + n * 4)$. Dokładniej o wektorach obsługi zdarzeń specjalnych (*exceptions*) będziemy mówili później.
Przerwanie programowe przy wystąpieniu nadmiaru `TRAPV` (*TRAP on oVerflow*) działa analogicznie, ale jego obsługa jest wykonywana tylko w przypadku ustawienia flagi nadmiaru w CCR:

- jeśli flaga nadmiaru ustawiona: $CCR : V = 1$
  - przełączenie w tryb *supervisor*: $1 \rightarrow SR : S$
  - wyskładowanie PC i SR na stosie wskazywanym przez SSP (*supervisor Stack Pointer*): $(SSP) - 4 \rightarrow SSP$; $(PC) \rightarrow (SSP)$; $(SSP) - 2 \rightarrow SSP$; $(SR) \rightarrow (SSP)$
  - załadowanie do PC adresu obsługi trapu: $(\$1C) \rightarrow PC$

Wektor onumerze 7 (adresie $\$1C$) jest przeznaczony do adresowania procedury obsługi przerwania programowego przy nadmiarze. Dzięki opisanej instrukcji można uprościć testowanie i obsługę nadmiaru w bibliotekach matematycznych (jedna procedura dla całej biblioteki, wywoływana z dowolnego poziomu zagłębienia podprogramów i do tego obsługiwana w trybie uprzywilejowanym). Aby uniknąć nieporozumień jeszcze raz zwróćmy uwagę, że aby wykorzystać opisane własności, instrukcja `TRAPV` **musi zostać użyta po każdej operacji arytmetycznej, która może dać nadmiar**.
Instrukcja `ILLEGAL` działa analogicznie jak opisane wcześniej przerwania programowe:

- przełączenie w tryb *supervisor*: $1 \rightarrow SR : S$
- wyskładowanie PC i SR na stosie wskazywanym przez SSP (*supervisor Stack Pointer*): $(SSP) - 4 \rightarrow SSP$; $(PC) \rightarrow (SSP)$; $(SSP) - 2 \rightarrow SSP$; $(SR) \rightarrow (SSP)$
- załadowanie do PC adresu obsługi instrukcji nielegalnej: $(\$10) \rightarrow PC$

W rzeczywistości mamy więcej niż jedną instrukcję nielegalną (to znaczy taki kod maszynowy, który w danym procesorze nie został sensownie wykorzystany). Każda z takich instrukcji powoduje opisane wyżej zachowanie procesora. W miarę rozwoju rodziny M68K instrukcji nielegalnych ubywa, lecz wspomniana wyżej (o kodzie `$4AFC`) pozostaje. Można ją wykorzystać do wypełnienia obszaru pamięci, w którym procesor nie powinien szukać kodu programu. Wtedy każdy błąd (zawiniony zazwyczaj przez programistę, choć być może i przez dekodowanie adresu pamięci) będzie wychwytywany i obsługiwany w przewidzianej do tego celu procedurze, której adres trzeba umieścić w wektorze 4 (adres `$10`).
Instrukcja `CHK` (*CHecK register against bounds*) pozwala sprawdzać, czy zawartość rejestru danych mieści się pomiędzy 0 a podaną w pierwszym argumencie górną granicą:

```
                D/A          Dn
               -d(A,X)+      ---          X N Z V C
CHK             abs     ,    ---          - ! ? ? ?
               d(PC,X)       ---
                imm          ---
```

Dolne słowo rejestru danych (począwszy od MC68020 można również sprawdzać dane 32-bitowe - `.L`) jest porównywane z zerem i górną granicą.
Gdy zawartość rejestru danych mieści się w podanych granicach, to wszystkie flagi (poza X, która pozostaje nienaruszona) są niezdefiniowane, a instrukcja `CHK` nie daje żadnego efektu.
Jeśli granice zostały przekroczone, to flaga N jest ustawiana, jeśli $(Dn) < 0$, a zerowana, gdy $(Dn) > (< ea >)$.
Dodatkowo następuje obsługa zdarzenia specjalnego (*exception*) według wektora 6:

- przełączenie w tryb *supervisor*: $1 \rightarrow SR : S$
- wyskładowanie PC i SR na stosie wskazywanym przez SSP (*supervisor Stack Pointer*): $(SSP) - 4 \rightarrow SSP$; $(PC) \rightarrow (SSP)$; $(SSP) - 2 \rightarrow SSP$; $(SR) \rightarrow (SSP)$
- załadowanie do PC adresu obsługi operacji `CHK`: ($18) $\rightarrow PC$

Procedura, której adres wpisano do wektora 6 (pod adres `$18`) może służyć do systemowej obsługi przekroczenia wielkości tablic, typów okrojonych itp.
Instrukcja pusta `NOP` (*No OPeration*) jest bezargumentowa, nie powoduje żadnych zmian w procesorze poza zwiększeniem licznika rozkazów, a jej kod zajmuje jedno słowo w pamęci. Jest to najszybsz instrukcja procesora, więc może służyć do odmierzania odcinków czasu w programowych pętlach opóźniających. Jest też czasem przydatna przy uruchamianiu oprogramowania na poziomie maszynowym, gdyż jej kodem można zamaskować inne (niepożądane) instrukcje w uruchamianym programie.
Kolejnym przykładem wykorzystania `NOP` jest prosty sprzętowy tester procesora. Wyobraźmy sobie, że wymusimy na magistrali danych procesora słowo odpowiadające kodowi instrukcji `NOP` (`$4E71`). Można to zrobić nawet przy pomocy odpowiednich zworek do masy i wysokiego poziomu logicznego. Procesor pobierając z magistrali kolejne instrukcje będzie zwiększał licznik rozkazów o 2 po każdym pobraniu, bo tyle bajtów zajmuje instrukcja `NOP`. W ten sposób kolejne linie adresowe procesora będą się zachowywały jak wyjścia licznika binarnego. Zwykły oscyloskop może posłużyć do sprawdzenia poprawności działania procesora i ewentualnych dekoderów adresowych w testowanym układzie.

2.3.6 Instrukcje uprzywilejowane

Pewne instrukcje mogą być wykonywane tylko wtedy, gdy procesor jest w trybie nadzorcy (*supervisor*). Jest to sygnalizowane stanem "1" flagi S w rejestrze statusowym. Do takich instrukcji należą:

- operacje na rejestrze statusowym: `MOVE SR,[.]`, `MOVE [.],SR`, `ANDI [#],SR`, `ORI [#],SR`, `EORI [#],SR`
- operacje na wskaźniku stosu użytkownika: `MOVE USP,An`, `MOVE An,USP`
- powrót z obsługi zdarzenia specjalnego: `RTE`
- instrukcje pomocnicze: `RESET`, `STOP`

Próba wykonanie którejkolwiek z powyższych instrukcji w chwili, gdy procesor jest w trybie użytkownika, powoduje zdarzenie specjalne naruszenia uprzywilejowania (*privillege violation*) obsługiwane według wektora 8:

- przełączenie w tryb *supervisor*: $1 \rightarrow SR : S$
- wyskładowanie PC i SR na stosie wskazywanym przez SSP (*supervisor Stack Pointer*): $(SSP) - 4 \rightarrow SSP$; $(PC) \rightarrow (SSP)$; $(SSP) - 2 \rightarrow SSP$; $(SR) \rightarrow (SSP)$
- załadowanie do PC adresu obsługi naruszenia uprzywilejowania: ($20) $\rightarrow PC$

Twórca systemu operacyjnego powinien zadbać o odpowiednią obsługę takiego przypadku (adres procedury obsługi tego zdarzenia winien się znaleźć w wektorze 8 (adres `$20`).
Operacje na rejestrze statusowym: `MOVE SR,[.]`, `MOVE [.],SR`, `ANDI [#],SR`, `ORI [#],SR`, `EORI [#],SR` są analogiczne do instrukcji operujących w trybie użytkownika (nieuprzywilejowanym) na dolnym bajcie SR (CCR). Istotne jest, że teraz rozmiar danych wynosi 16 bitów:

```
* zerowanie I[0..2] (maski przerwan):
            ANDI.W  #%1111100011111111,SR   bity 8-10 w SR
* ustawianie I[0..2] na 5
            ORI.W   #%10100000000,SR        maskowanie poziomow <=5
* zmiana flagi T na przeciwna wartosc:
            EORI.W  #%1000000000000000,SR   bit 15 w SR
```

Pewną niekonsekwencją jest, że w procesorach MC68000 i MC68008 (wersja 68000 z 8-bitową magistralą danych) instrukcja `MOVE SR,[.]` nie jest uprzywilejowana. Dopiero poczynając od MC68010 ujednolicono uprzywilejowanie dostępu do rejestru statusowego.
Operacje na wskaźniku stosu użytkownika: `MOVE USP,An`, `MOVE An,USP` pozwalają w trybie systemowym zarówno czytać, jak i modyfikować zawartość rejestru wskaźnika stosu użytkownika (USP – *User Stack Pointer*). Operacje te nie naruszają flag, a pozwalają między innymi na zmianę wskaźnika stosu przy zmianie użytkownika (przełączanie kontekstu w wielozadaniowych systemach operacyjnych). pamiętajmy, że odwołanie do wskaźnika stosu poprzez A7 daje w trybie systemowym dostęp do wskaźnika stosu systemowego (SSP – *Supervisor Stack Pointer*).
Instrukcja `RTE` (*ReTurn from Exception*) umożliwia powrót z obsługi zdarzenia specjalnego. W procesorach MC68000 i MC68008 jej wykonanie powoduje:

- odtworzenie SR i PC ze stosu wskazywanego przez SSP (*supervisor Stack Pointer*): $((SSP)) \rightarrow SR$; $(SSP)+2 \rightarrow SSP$; $((SSP)) \rightarrow PC$; $(SSP)+4 \rightarrow SSP$

Zwróćmy uwagę, że zazwyczaj instrukcja ta kończy procedurę obsługi zdarzenia specjalnego. Jeśli jego wywołanie nastąpiło w trybie użytkownika (SR:S=0), to odtwarzanie SR ze stosu spowoduje wyjście z trybu systemowego. W procesorach od MC68010 wzwyż obsługa zdarzeń specjalnych składuje na stosie systemowym więcej informacji w postaci tzw. ramek stosu, których format jest pamiętany w słowie na stosie (różne zdarzenia specjalne mają różne formaty ramek). Wtedy instrukcja `RTE` usuwa ze stosu również te dodatkowe informacje, zgodnie z odczytanym ze stosu formatem.
Instrukcja `STOP` zatrzymuje pracę procesora do czasu przerwania sprzętowego lub restartu. Ma jeden argument natychmiastowy i nie narusza flag:

```
                                                X N Z V C
     STOP          imm                          - - - - -
```

Argument ten stanowi nową zawartość rejestru statusowego:

- ustawienie SR na wartość argumentu: $< imm > \rightarrow SR$
- zatrzymanie procesora

Zauważmy, że przy obsłudze wejścia/wyjścia może się zdarzyć, że do czasu przyjścia nowych danych procesor nie ma żadnych zadań. Jeśli fakt przyjścia danych jest sygnalizowany przerwaniem o poziomie np. 6, to możemy zatrzymać procesor do czasu przyjścia takiego przerwania przez zamaskowanie (w SR:I[0..2]) wszystkich przerwań o niższych poziomach:

```
STOP #%0010010100000000 bit S=1 i I[2..0]=%101
```

Takie zatrzymanie procesora jest szczególnie cenne, jeśli zależy nam na oszczędzaniu źródła zasilania (np. baterii) w systemach autonomicznych. Nowe wersje procesorów i kontrolerów Motoroli są wykonane w technologii HCMOS (*High speed CMOS*), która pozwala drastycznie obniżyć pobór mocy ze źródla zasilającego przy zatrzymaniu pracy procesora.
W procesorach M68K linia RESET jest dwukierunkowa. Wymuszenie na niej stanu niskiego z zewnątrz powoduje zainicjowanie pracy procesora **po przywróceniu stanu nieaktywnego (wysoki poziom napięciowy)**. Z drugiej strony, procesor może wymusić sygnał restartu na linii RESET w celu zainicjowania sprzętowych układów towarzyszących (porty, przerzutniki itp.). Instrukcja `RESET` pozwala programowo wywołać stan aktywny (niski poziom napięciowy) na tej linii.
Instrukcja `RESET` **nie powoduje żadnych zmian w procesorze**. Powinna być stosowana na początku procedury inicjacji systemu. Jej użycie w innym czasie, w trakcie normalnej pracy prawdopodobnie spowoduje niechciane objawy, ze względu na utratę inicjacji przez porty i inne elementy otoczenia procesora.

3 Łączenie asemblera z językami wyższego rzędu

Aby wykorzystać możliwości asemblera i języka wyższego rzędu (C), trzeba znać podstawowe zasady konstruowania programów, które są stosowane w kompilatorach. Nie są one jednakowe dla wszystkich kompilatorów. Stosowane konwencje zależą od wybranego procesora (w naszym przypadku M68K) i środowiska, w którym ma pracować wynikowy program (program samodzielny, współpracujący z monitorem/debuggerem, uruchamiany jako proces przez system wielozadaniowy). Różni producenci oprogramowania w różny sposób wykorzystują zasoby komputera (rejestry procesora, pamięć). Poniżej podano opis konwencji stosowanych w pakiecie oprogramowania skrośnego firmy Boston Systems Office/Tasking ([7]). Pakiet ten składa się z asemblera, kompilatora języka C, linkera i formatera. Pracuje w środowisku MS–DOS i umożliwia tworzenie oprogramowania dla procesorów z rodziny M68K.

3.1 Segmentowa budowa programów

Programy (poza najprostszymi przykładami) są zazwyczaj pisane w postaci modułów zawierających poszczególne funkcje i procedury, definicje danych, deklaracje zmiennych itp. Każdy z modułów winien stanowić możliwie zamkniętą całość. W jego skład powinny wchodzić zarówno rezerwacje obszarów pamięci na zmienne, jak i kod programu. Kompilator lub asembler tworzą z takiego źródła wynikowy moduł relokowalny. Zawiera on kod maszynowy dla procesora, wymaga jednak modyfikacji w miejscach zawierających adresy procedur i zmiennych oraz inne stałe wartości zdefiniowane w innych modułach. Nazwy symboliczne, którym w danym module nie można przypisać wartości, są deklarowane jako zewnętrzne. W języku C służy do tego celu dyrektywa `#extern`, a w asemblerze – dyrektywa `XREF` (*eXternal REFerence*). Do oznaczenia nazw, które są zdefiniowane w module asemblerowym, a przeznaczone do publicznego wykorzystania, służy dyrektywa `XDEF` (*eXternal DEFinition*).
Obszar pamięci zajmowany przez elementy składowe modułu jest reprezentowany przez **segmenty**. Segment stanowi ciąg bajtów o kolejnych adresach, które mogą być umieszczone pod pewnym adresem bezwzględnym. W programach asemblerowych segmenty definiuje się bezpośrednio (dyrektywą `section`), a w programach w języku C – pośrednio (kompilator definiuje segmenty według własnych konwencji). Każdy segment ma nazwę, długość i inne atrybuty. Początkową zawartość segmentu może stanowić kod maszynowy lub dane (nie wszystkie dane w segmencie muszą być zainicjowane). Rozdzielenie zmiennych od programu umożliwia np. umieszczenie kodu w obszarze pamięci ROM, a zmiennych w RAM.
Łączenie modułów wykonuje linker/lokator. Jest to program, któremu podaje się listę modułów do połączenia, biblioteki do przeszukiwania oraz informację o obszarach pamięci, które należy wykorzystać dla poszczególnych segmentów. Proces łączenia kilku modułów relokowalnych w bezwzględny moduł wynikowy polega na układaniu segmentów kolejnych modułów relokowalnych w zadanych obszarach pamięci. W różnych modułach mogą występować segmenty tego samego typu (np. segment kodu). Są one dokładane na końcu dotychczas zajętego obszaru w tym segmencie (ewentualnie z zachowaniem parzystości adresów). Przypisanie wszystkim segmentom danego modułu bezwzględnych adresów powoduje, ze publiczne nazwy zdefiniowane w tym module otrzymują bezwzględne wartości. Tymi wartościami są uzupełniane zewnętrzne odwołania innych modułów. W miarę potrzeby mogą być dołączane moduły z bibliotek, które są przeszukiwane na końcu, gdy jakieś nazwy pozostają niezdefiniowane.
Specjalnym typem segmentów są segmenty oddzielne (*separate segments*). Dla kompilatora C deklaruje się je dyrektywą `#option separate`. Zmienne umieszczone w segmentach tego typu mogą być umieszczane przez linker w zadanym, bezwzględnie adresowanym miejscu. Można to wykorzystać np. do obsługi portu bezpośrednio z programu w języku C:

```
#option separate port
char port;

char czytaj_port(void)
{
return(port);
}
```

Kompilator umieści zmienną `port` w oddzielnym segmencie `S_port`. Aby funkcja *czytaj_port* zracała wartość odczytaną z rejestru o adresie $fffe00, wystarczy w opisie rozmieszczenia segmentów dla linkera podać:

```
LOCATE ( S_port : #fffe00 );
```

3.1.1 Typy segmentów

Kompilator tworzy różne segmenty dla kodu i danych. Ich nazwy i zawartość zestawiono w tabeli:

segment	zawartość
S_*fname*	segment kodu dla modułu, którego pierwsza funkcja ma nazwę *fname*
S_*vname*	oddzielny segment danych dla zmiennej *vname*
idata	dane globalne inicjowane
udata	dane globalne nieinicjowane
sdata	stałe tekstowe
cdata	dane stałe (dyrektywa `#const`)
libcode	kod asemblerowych funkcji bibliotecznych
init	procedura startowa `__main`
init@0	wektory SSP i PC pod adresem 0
BREAKSEG	procedura obsługi pułapek dla debuggera
BREAKSEG@16	wektor obsługi `TRAP` pod adresem $10

Grupa (*group*) jest zestawem segmentów, które muszą być umieszczone w obszarze pamięci nie przekraczającym 64K. Grupa `data` generowana przez kompilator zawiera segmenty `idata` (w którym są umieszczane wszystkie zmienne globalne, które mają nadawaną wartość początkową) i `udata` (w którym są umieszczane zmienne globalne bez początkowej wartości).
Klasa (*class*) stanowi nazwany zestaw segmentów mających wspólną własność. Nie ma ona wpływu na generację kodu (jak to było w przypadku grupy), a może służyć do wygodniejszego definiowania obszarów dla lokatora (dyrektywa `LOCATE`). Nazwy klas i ich zawartość zestawiono w tabeli:

klasa	zawartość
{code}	kod (poza kodem funkcji bibliotecznych)
{data}	dane globalne (poza oddzielnymi segmentami)
{isep}	inicjowane dane w oddzielnych segmentach
{usep}	nieinicjowane dane w oddzielnych segmentach
{stsep}	statyczne dane w oddzielnych segmentach
{separate}	dane w oddzielnych segmentach nazwanych przez użytkownika
{}	(klasa pusta – *null*) kod asemblerowy, w tym funkcje biblioteczne

Nie wszystkie segmenty są relokowalne. Asembler może generowaś segmenty bezwzględne (*absolute segments*). Dzieje się tak np. przy zastosowaniu dyrektywy `ORG`. Takie segmenty są umieszczane przez linker/lokator w pierwszej kolejności, by uniknąć kolizji adresów bezwzględnych. W drugiej kolejności wykonywane są dyrektywy podane przez użytkownika, a następnie (dla pozostałych segmentów) stosowany jest domyślny algorytm rozmieszczania.

Dostępne komeny lokatora:

`DECLARE`	definicja nieokreślonej nazwy zewnętrznej `DECLARE (nazwa : adres);`
`LOCATE`	specyfikacja umieszczenia segmentu (klasy) `LOCATE (lista-nazw : zakres-adresow);`
`MEMORY`	specyfikacja wielkości pamięci `MEMORY (adres);`
`RESERVE`	rezerwacja obszaru pamięci `RESERVE (zakres-adresow);`
`SEGSIZE`	powiększanie wielkości segmentu `SEGSIZE (nazwa : liczba);`
`START`	specyfikacja adresu startowego `START (adres);`

nazwa dotyczy segmentu, klasy lub symbolu globalnego; nazwy klas podaje się w nawiasach klamrowych (np. `{data}`) lub poprzedza kwalifikatorem `CLASS` (np. `CLASS (data)`); pusta para nawiasów (`{}`) oznacza klasę pustą;

lista-nazw jest ciągiem nazw oddzielonych białymi znakami;

liczba jest podawana dziesiętnie, lub szesnastkowo (z przedrostkiem `#`);

adres jest podawany jako liczba;

zakres-adresow może być podany od–do (`#100 TO #1000`), ograniczony od góry (`BEFORE #1000`) lub od dołu (`AFTER #10000`); górny adres nie jest wliczany do zakresu;

3.2 Konwencje kompilatora C

Przy łączeniu procedur pisanych w języku asemblera z programami pisanymi w C nieodzowna jest znajomość konwencji stosowanych przez kompilator języka C przy tworzeniu kodu. Podstawowe elementy wymagające omówienia to: zasady przydzielania pamięci na zmienne, sposoby korzystania z rejestrów, metody przekazywania parametrów wywołania dla proceduri funkcji, sposób zwracania wyników przez funkcje, obsługa stosu, inicjacja programu przy starcie.

3.2.1 Przydział pamięci

Typy danych są reprezentowane w języku maszynowym w postaci słów o różnych rozmiarach. Opisywany kompilator wytwarza kod, w którym te rozmiary są następujące:

`char`	8 bitów	bez znaku
`short`	8 bitów	ze znakiem
`int`	16 bitów	ze znakiem
`unsigned`	16 bitów	bez znaku
`long`	32 bity	ze znakiem
`pointer`	32 bity	adres bezwzględny

Użytkownik może zmienić sposób reprezentacji niektórych typów przy pomocy opcji `-D` przy wywołaniu kompilatora. Argumenty tej opcji mają postać: *tbs*, gdzie *t* oznacza typ danych, *b* – ilość bajtów, a *s* – atrybut znaku (z lub bez znaku). Typy są oznaczane przez: `c - char`, `i - int`, `s - short`, ilość bajtów jest podawana jako cyfra, a atrybut znaku może przyjmować wartość `s - signed` lub `u - unsigned`. Dopuszczalne są kombinacje: `c1s, c1u, i2s, i4s, s1s, s2s`, z których domyślne są: `c1u, i2s, s1s`.
Trzeba pamiętać, by wszystkie moduły programu były kompilowane z jednakowym ustawieniem parametrów opcji `-D`.

W wyniku kompilacji programu napisanego w języku C zmienne użytkownika mogą być umieszczone w różnych obszarach:

- na stosie
- w rejestrach `A1-A3/D2-D7`
- w obszarze danych globalnych wskazywanym przez `(A5)`
- w oddzielnych segmentach danych (`#option separate`)

Zmienne lokalne, zadeklarowane w procedurach, są umieszczane na stosie. Wyjątek stanowi 9 pierwszych zmiennych zadeklarowanych jako `register`. Są one umieszczane w rejestrach `A1-A3` (wskaźniki) i `D2-D7` (zmienne innych typów). Zmienne na stosie są adresowane względem wskaźnika ramki stosu (`A6`) z 16–bitowym przesunięciem. Wskaźnik ramki stosu jest inicjowany rozkazem `LINK A6,#-n` na początku procedury. Dodatnie przesunięcia pozwalają dotrzeć do parametrów wywołania procedury, a ujemne – do zmiennych lokalnych. Obszar zmiennych lokalnych jest ograniczony do 32K bajtów ($2^{15} = 32768$).
Zmienne globalne są umieszczane w obszarze danych globalnych. Dla efektywnego adresowania tych zmiennych zastosowano pośredni tryb rejestrowy względem `A5` z 16–bitowym przesunięciem. Ogranicza to całkowity rozmiar danych globalnych do 64K bajtów. Przekroczenie tego limitu jest wykrywane dopiero w fazie łączenia (*linkowania*) programu.
Zmienne zadeklarowane przy pomocy dyrektywy `#option separate` są umieszczane w oddzielnych segmentach danych. Dla każdego takiego segmentu kompilator tworzy oddzielny (4–bajtowy) wskaźnik umieszczany w obszarze danych globalnych i inicjowany adresem obszaru pamięci przydzielonego na segment. Adres ten jest wykorzystyany przy każdym dostępie do zmiennych z tego segmentu. Ten sposób dostępu do zmiennych jest mniej efektywny, ale nie ogranicza wielkości obszaru danych. Trzeba pamiętać, że przy definiowaniu w module pisanym w języku asemblera danych, które mają być dostępne w modułach pisanych w języku C jako zewnętrzne zmienne typu `separate`, konieczne jest utworzenie i zainicjowanie wskaźnika na odpowiedni obszar, umieszczonego w obszarze danych globalnych.

3.2.2 Funkcje rejestrów

Rejestry procesora M68K są wykorzystywane przez kompilator następująco:

A1–A3	zmienne rejestrowe typu wsaźnikowego
D2–D7	zmienne rejestrowe innych typów
A5	wskaźnik obszaru danych globalnych
A6	wskaźnik ramki stosu
A7	wskaźnik stosu
A0	wartość zwracana przez funkcję (typu wskaźnikowego)
D0	wartość zwracana przez funkcję (innych typów)

Każda procedura musi zachowywać wartości rejestrów D2–D7, A1–A3 i A5–A7.

3.2.3 Przekazywanie parametrów

Parametry wywołania procedur są przekazywane za pośrednictwem stosu. Jeśli parametry mają rozmiar pojedynczego bajtu, to są składowane na stosie w postaci słowa 16–bitowego, którego starszy bajt nie jest używany. Kod tworzony przez kompilator w celu wywołania procedury składa się z następujących kroków:

1. Umieszczenie argumentów na stosie, poczynając od ostatniego, na pierwszym kończąc.
2. Wywołanie funkcji (podprogramu).
3. Usunięcie argumentów ze stosu.

3.2.4 Prolog i epilog procedury

W celu zapewnienia dopasowania procedury napisanej w języku asemblera do opisanych wyżej zasad tworzenia kodu przez kompilator języka C należy ją wyposażyć w prolog i epilog.
Prolog składa się z dwóch instrukcji:

```
LINK    A6,#-n              inicjacja ramki stosu (n bajtow)
MOVEM.L <lista rej.>,-(A7)  zachowanie uzywanych rejestrow
```

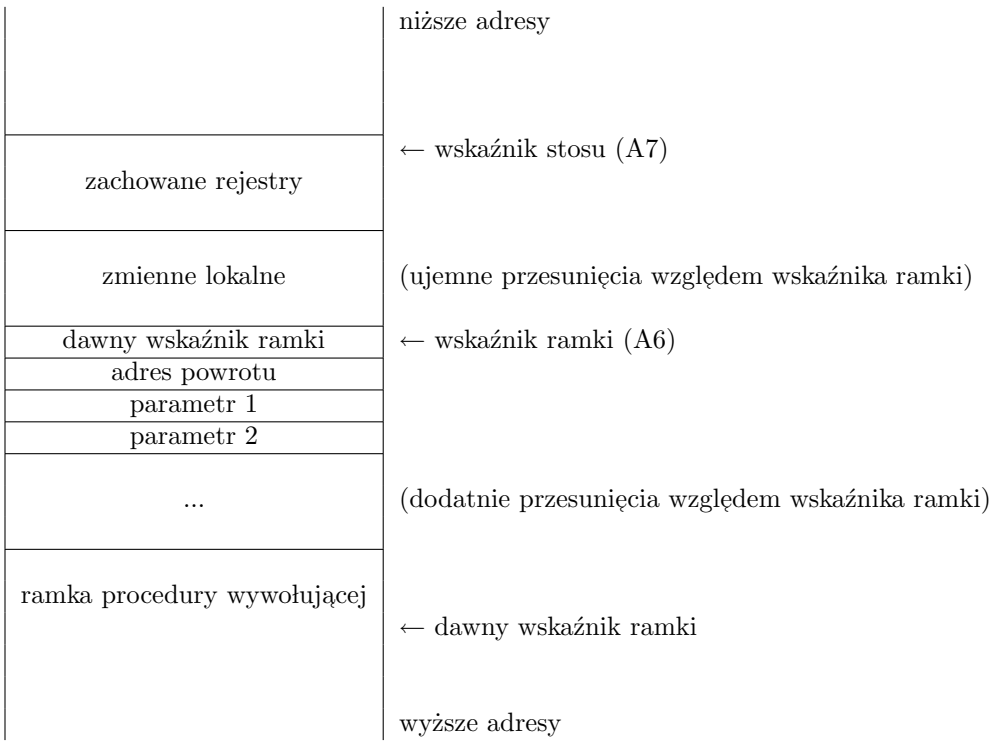
Epilog odwraca działanie prologu i powoduje zakończenie podprogramu:

```
MOVEM.L (A7)+,<lista rej.>  odtworzenie uzywanych rejestrow
UNLK    A6                  usuniecie ramki stosu
RTS                         powrot z procedury
```

Lista rejestrów w instrukcjach `MOVEM` musi zawierać wszystkie uzywane wewnątrz procedury rejestry spośród A1–A3 i D2–D7.
Nie należy wewnątrz procedury zmieniać zawartości rejestrów A5 i A6. Względem A5 odwołujemy się w całym programie do zmiennych globalnych, a względem A6 – do parametrów wywołania procedury i jej zmiennych lokalnych umieszczonych na stosie.
Nie zachowujemy zawartości rejestrów D0 i A0, ponieważ w jednym z nich ma pozostać wynik funkcji w celu udostępnienia go procedurze, z której wywoływano procedurę bieżącą.

3.2.5 Budowa ramki stosu dla procedury

Typowa ramka stosu dla procedury składa się z parametrów wywołania, umieszczanych (a po powrocie z podprogramu usuwanych) przez procedurę wywołującą, zachowanego wskaźnika ramki stosu procedury wywołującej (wynik instrukcji `LINK` z prologu procedury), zarezerwowanego miejsca na lokalne zmienne procedury (wynik instrukcji `LINK`). Wskaźnik stosu jest ustawony na koniec ramki stosu, co pozwala na dalsze zagłębianie wywołań. W przypadku wykorzystywania rejestrów wewnątrz procedury, ich dotychczasowa zawartość jest składowana na stosie.



Wykonanie epilogu powoduje przywrócenie zawartości rejestrów ze stosu i likwidację ramki stosu (instrukcja `UNLK` przestawia wskaźnik stosu (A7) zgodnie ze wskaźnikiem ramki (A6) i odtwarza dawny wskaźnik ramki ze stosu. Powrót z podprogramu (`RTS`) przywraca ze stosu licznik rozkazów schowany podczas wywołania podprogramu i pozwala na kontynuację procedury wywołującej. Do jej obowiązków należy usunięcie ze stosu umieszczonych tam wcześniej parametrów wywołania.

3.2.6 Procedura startowa programu

Program napisany w języku C musi zawierać funkcję główną (*main*). Zgodnie z konwencją tworzenia nazw przez kompilator, punkt startowy tej funkcji otrzymuje nazwę `_main`.

```
    xref    _main
    xref    data

    section init,,''code''
    xdef    __main
__main

    move    SR,D0
    btst    #13,D0          Test bitu trybu (SR:S)
    beq.s   ustate          Pomin, gdy w trybie uzykownika
    move.l  #$00007ffc,D1   Ustaw SSP (A7') na adres 00007ffc
    movec   D1,SSP
    and     #$d8ff,D0       Wylacz bit ''S'' i bity ''I0-I2''
    move    D0,SR           Przejdz do trybu uzytkownika z odblokowanymi
*                           przerwaniami
ustate
    movea.l #$00007f00,A7   Ustaw USP (A7) na adres 00007f00
    lea     data,A5         Zaladuj do A5 adres obszaru danych globalnych
    suba.l  A6,A6           A6 - wskaznik ramki stosu (0 oznacza szczyt)

*                           Zeruj rejestry (dla debuggera)
    suba.l  A4,A4
    suba.l  A3,A3
    suba.l  A2,A2
    suba.l  A1,A1
    suba.l  A0,A0
    clr.l   D7
    clr.l   D6
    clr.l   D5
    clr.l   D4
    clr.l   D3
    clr.l   D2
    clr.l   D1
    clr.l   D0

    jsr _main               Wywolaj glowna procedure programu

loop
    jmp     loop            Po powrocie z _main - martwa petla

*   init    ends

* Tu sa zdefiniowane wektory restartu (segment absolutny!)
* dla wersji samodzielnej programu.
```

```
* Pod adresem 0 jest startowy  SSP, a pod adresem 4 - startowy PC.

    org 0
    dc.l    $00007ffc
    dc.l    __main

* Etykieta przy dyrektywie ''end'' oznacza adres startowy ''.start''
* uzywany do generacji rekordu S9 w przypadku wersji z monitorem.
    end     __main
```

Uruchomienie programu napisanego w języku C wymaga pewnych przygotowań przed wywołaniem procedury `_main`. Najważniejsze z nich to zainicjowanie początkowej ramki stosu (rejestr A6) i wskaźnika obszaru zmiennych globalnych (rejestr A5).
Po zakończeniu funkcji *main* (powrót z podprogramu) musi nastąpić właściwe zakończenie (przekazanie sterowania do systemu operacyjnego, monitora/debuggera, lub – w najprostszym przypadku – zatrzymanie procesora w nieskończonej pętli).
Wymienione zadania realizuje procedura startowa `__main` umieszczona w bibliotece. Kompilator tworzy odwołanie do nazwy `__main`, co powoduje automatyczne dołączenie procedury startowej przez linker. Punktem startowym kodu wynikowego jest etykieta `__main`.
W przykładowej martwej pętli po powrocie z `_main` nie jest analizowana wartość wynikowa *n*, którą może zwrócić funkcja *main* przez *return n*. W konkretnym środowisku (system operacyjny, monitor/debugger) należy z tego miejsca powrócić do systemu.
Poza powrotem z funkcji głównej programu napisanego w języku C przez `RTS`, co odpowiada zakończeniu funkcji *main* przez *return*, do zakończenia programu używana jest funkcja *exit*. Jej najprostsza implementacja polega na przejściu do niekończącej się pętli:

```
    section libcode,,''code''
    xdef    _exit
    xdef    __exit
_exit:
__exit:
*
* Najprostsze wyjscie - martwa petla
*
    jmp     __exit
    end
```

Procedura `_exit` powinna zamykać wszystkie otwarte pliki i wywoływać procedure `__exit`. W powyższym przykładzie obie funkcje są trywialne. Pominięto również analizę kodu wyjścia, który może być podany jako argument funkcji *exit()*. Tak jak poprzednio, modyfikacje zależą od środowiska.

3.3 Przykład

Dla zilustrowania omówionych zasd łączenia modułów napisanych w języku asemblera z modułami napisanymi w języku C rozważmy przykładowy program *demo* wypisujący na wyświetlaczu LCD napis: "Test LCD (MW)". Główna funkcja programu (*main()*) i funkcja wysyłania pojedynczego znaku na LCD (*putdata()*) są napisane w C.

```
/*************************************************************************
*
* demo.c    (MW)
*
* Przykladowy program do demonstracji laczenia modulow w C i asemblerze
*
*************************************************************************/

#include <stdio.h>

#define LCD_IR 0xeff800
#define LCD_DR 0xeff801

char * LcdIr = (char *) LCD_IR;
char * LcdDr = (char *) LCD_DR;

extern void initlcd(void);

void putdata(char data)
{
    while((*LcdIr & 0x80) == 0x80);
    *LcdDr = data;
}

main()
{
    char * ptr = ''Test LCD (MW)\0'';

    initlcd();  /* inicjacja wyswietlacza */

    while(*ptr)
    {
        putdata(*ptr++);
    }
}

/* koniec demo.c */
```

Inicjacja wyświetlacza (*initlcd()*) jest napisana w asemblerze i dołączona jako funkcja zewnętrzna.

```
*************************************************************************
*
* initlcd.asm   (MW)
*
* Przykladowa funkcja do demonstracji laczenia modulow w C i asemblerze
*
*************************************************************************

    section S_initlcd,,''code''

LCDIR   equ $eff800

    xdef    _initlcd

_initlcd
    link    a6,#0

    move.b  #$38,d1
    bsr     putctrl
    move.b  #$08,d1
    bsr     putctrl
    move.b  #$06,d1
    bsr     putctrl
    move.b  #$80,d1
    bsr     putctrl
    move.b  #$0f,d1
    bsr     putctrl

    unlk    a6
    rts

putctrl
    btst    #7,LCDIR
    beq     putctrl
    move.b  d1,LCDIR
    rts

    end

* koniec initlcd.asm
```

W celu ułatwienia analizy powiązań pomiędzy modułami w C i w asemblerze zamieszczono wydruk asemblerowej postaci programu wyprodukowany przez kompilator z odpowiednimi liniami źródłowymi umieszczonymi w komentarzach.

```
    SECTION idata,,''data''
    XDEF    _LcdIr
_LcdIr  DC.L    15726592
    XDEF    _LcdDr
_LcdDr  DC.L    15726593

    SECTION S_putdata,,''code''

*1
*2  /*****************************************************************************
*3  *
*4  * demo.c    (MW)
*5  *
*6  * Przykladowy program do demonstracji laczenia modulow w C i asemblerze
*7  *
*8  *****************************************************************************/
*9
*10 #include <stdio.h>
*11
*12 #define LCD_IR 0xeff800
*13 #define LCD_DR 0xeff801
*14
*15 char * LcdIr = (char *) LCD_IR;
*16 char * LcdDr = (char *) LCD_DR;
*17
*18 extern void initlcd(void);
*19
*20 void putdata(char data)
    XDEF    _putdata
_putdata
    LINK    A6,#0
__P1    EQU $000004

L20001
*21 {
*22     while((*LcdIr & 0x80) == 0x80);
    MOVEA.L _LcdIr-data(A5),A4
    MOVE.B  (A4),D1
    ANDI    #128,D1
    CMPI    #128,D1
    BEQ.S   L20001

*23     *LcdDr = data;
    MOVEA.L _LcdDr-data(A5),A0
    MOVE.B  9(A6),(A0)
    UNLK    A6
    RTS

* Function size = 32
```

```
    XREF    __main
* Variable ptr is in the A1 Register

*24 }
*25
*26 main()
    XDEF    _main
_main
    LINK    A6,#0
__P2    EQU $000024

*27 {
*28     char * ptr = ''Test LCD (MW)\0'';
    MOVEA.L #_N10,A1

*29
*30     initlcd();  /* inicjacja wyswietlacza */
    JSR _initlcd

*31
*32     while(*ptr)
    BRA L20002
L20003
*33     {
*34         putdata(*ptr++);
    CLR.L   D1
    MOVE.B  (A1)+,D1
    MOVE    D1,-(A7)
    JSR _putdata
    ADDQ.L  #2,A7
L20002
*(see line 32)
    TST.B   (A1)
    BNE.S   L20003
    UNLK    A6
    RTS

* Function size = 42
```

```
    SECTION sdata,,''constant''
_N10    DC.B    'Test LCD (MW)',0,0
* bytes of code = 74
* bytes of idata = 8
* bytes of udata = 0
* bytes of sdata = 15
    XREF    _initlcd
    XREF    data
    END
*35     }
*36 }
*37
*38 /* koniec demo.c */
```

Po zadaniu linkerowi następujących komend:

```
MEMORY ( #ffff );
LOCATE ( {code} {} : after #8000 );
LOCATE ( {constant} : after #8000 );
LOCATE ( {data} {usep} {isep} : after #3000 );
```

otrzymano wynikowy moduł bezwzględny, którego mapa pamięci (wyprowadzana na podstawie wyników pracy linkera przez pomocniczy program mmap) wygląda następująco:

```
Global       Address

_LcdIr       00003000 (12288)
_LcdDr       00003004 (12292)
__bufMax     00003008 (12296)
__buffer     0000300a (12298)
__main       00008000 (32768)
BREAKPT      00008060 (32864)
__end_       00008064 (32868)
_initlcd     00008066 (32870)
_putdata     000080a8 (32936)
_main        000080c8 (32968)

Group        Size Limit           Align  Member Segments

data         000009 (9)           hword  idata        udata

Segment      Address              Length           Class      Align  Combine

init@0       00000000 (0)         000008 (8)       <null>     hword  private
BREAKSEG@16  00000010 (16)        000004 (4)       <null>     lword  private
idata        00003000 (12288)     000008 (8)       data       hword  private
udata        00003008 (12296)     000001 (1)       data       hword  private
S__buffer    0000300a (12298)     0000c8 (200)     usep       hword  private
PSCT         00008000 (32768)     000000 (0)       <null>     hword  private
init         00008000 (32768)     00004a (74)      code       hword  private
BREAKSEG     0000804c (32844)     000018 (24)      code       lword  private
S__end_      00008064 (32868)     000002 (2)       code       hword  private
S_initlcd    00008066 (32870)     000042 (66)      code       hword  private
S_putdata    000080a8 (32936)     00004a (74)      code       hword  private
sdata        000080f2 (33010)     00000f (15)      constant   hword  private

Statistics

Segments   : 12
Globals    : 11
Groups     : 1
Code  Size : 000000f0 (240)
Data  Size : 00000009 (9)

User Start Address = #8000
```

Bibliografia

[1] Bennett J. M., *68000 Assembly Language Programming – A Structured Approach*, Prentice–Hall Inc., Englewood Cliffs, NJ, 1986.

[2] Greenfield J. D., *Microprocessor Handbook*, Wiley, 1985

[3] Harman T. L., Lawson B., *The Motorola MC68000 Microprocessor Familly: Assembly Language, Interface Design and System Design*, Prentice Hall, 1985

[4] King T., Knight B., *Programming the M68000*, Addison–Wesley, 1985

[5] Kostrzewski J., *Motorola 68000 - lista rozkazów mikroprocesora*, Elektronik, 1988.

[6] *M68000 Family Programmer's Reference Manual*, M68000PM/AD REV.1, Motorola Inc., 1992.

[7] *TaskTools C–68XXX compiler package for MS–DOS*, BSO Tasking, 1991.

[8] Triebel W. A., Singh A., *The 68000 Microprocessor*, Prentice Hall, 1985.

[9] Wnuk M., *OS–9 – modułowy, wielozadaniowy system czasu rzeczywistego*, Raport ICT SPR 31/94, Wydawnictwo Politechniki Wrocławskiej, Wrocław, 1994.